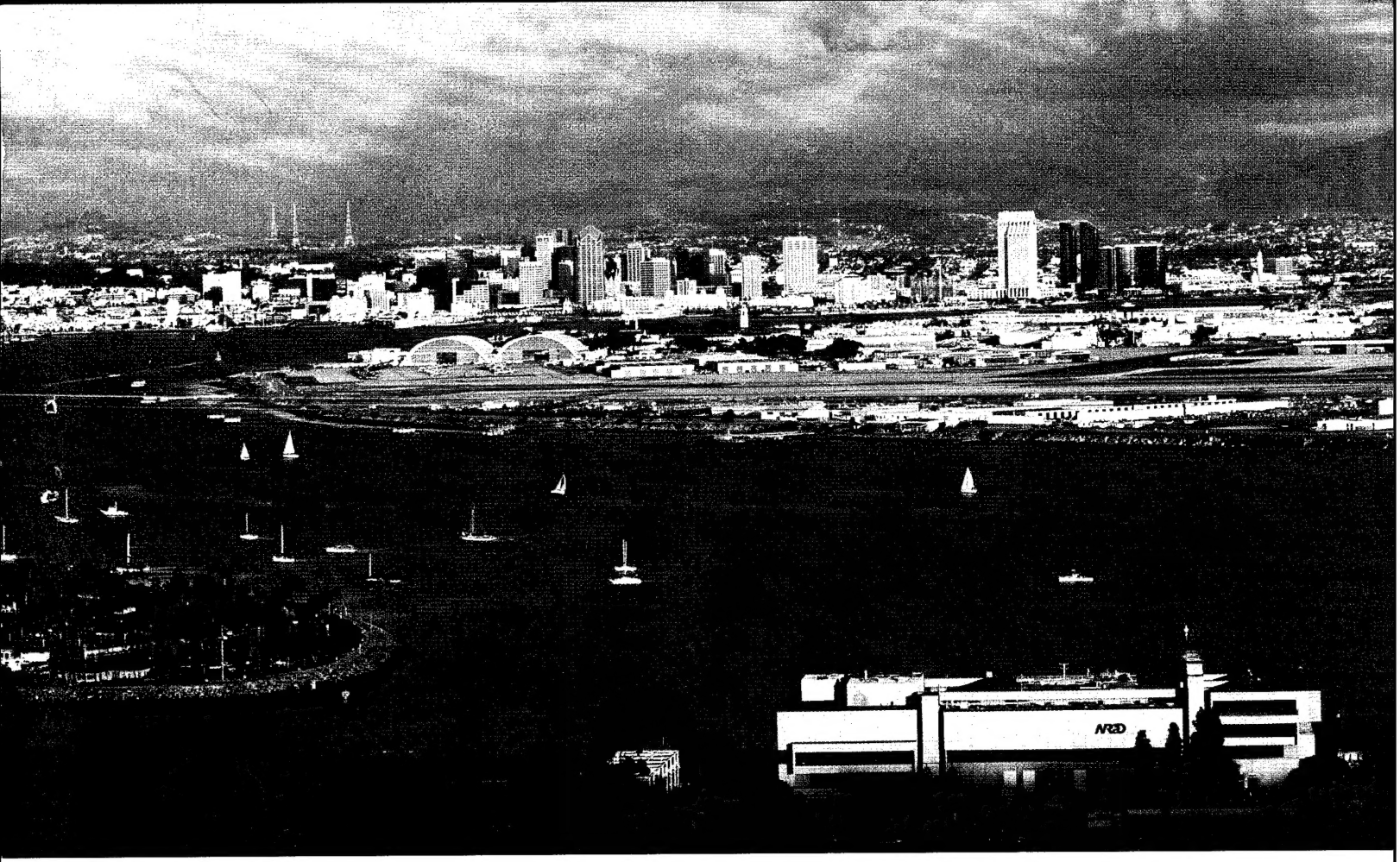


Scalable Programming Environment

Perry Partow Dennis Cottel

Technical Report 1672, Rev. 1
September 1995

Naval Command, Control and
Ocean Surveillance Center
RDT&E Division

San Diego, CA
92152-5001

Sponsored by Office of Naval Research

Technical Report 1672, Rev. 1
September 1995

Scalable Programming Environment

Perry Partow
Dennis Cottel

Sponsored by Office of Naval Research

NAVAL COMMAND, CONTROL AND OCEAN SURVEILLANCE CENTER
RDT&E DIVISION
San Diego, California 92152-5001

K. E. EVANS, CAPT, USN
Commanding Officer

R. T. SHEARER
Executive Director

ADMINISTRATIVE INFORMATION

The work reported here was performed in FY 1994 and FY 1995 under the Bottom Limited Active Classification project. The work was sponsored by the Office of Naval Research (ONR-321), 800 North Quincy Street, Arlington, VA 22217–5660, under program element 0602314N. This work was supported in part by a grant of High Performance Computer (HPC) time from the DoD HPC Distributed Center, Naval Command, Control and Ocean Surveillance Center (NCCOSC), RDT&E Division.

Released by
R. A. Dukelow, Head
Systems Design Branch

Under authority of
P. M. Reeves, Head
Analysis and Simulation Division

JA

EXECUTIVE SUMMARY

OBJECTIVE

This report describes the Scalable Programming Environment (SPE), a software tool that supports the development of scalable high-performance data-flow applications.

APPROACH

The SPE was developed on the Intel Paragon to support the Hybrid Digital Optical Processor (HyDOP) and the Bottom Limited Active Classification (BLAC) projects, both sponsored by the Office of Naval Research (ONR-321), for undersea surveillance acoustic signal processing. All the scalable and reconfigurable needs of these projects have been incorporated in a library of general programming calls. The development and testing of the SPE evolved as the these projects dictated. The Intel Paragon was made available by the DoD High Performance Computer (HPC) Modernization Program.

The SPE was designed with generality in mind, so that in addition to meeting the needs of acoustic signal processing, it could be used in other similar types of applications. The SPE has now been used for several synthetic aperture radar (SAR) processing applications, and this work has also influenced SPE development.

RESULTS

The SPE, which has been designed primarily to support data-flow processing applications, allows programs to be scaled to execute on any number of processing nodes while requiring no changes to the compiled binary code. The user is provided with a set of high-level message-passing routines that can be used to connect multi-instanced heterogeneous programs in a system. The SPE library routines hide the intricacies of how the parallel programs communicate. The details of the connections are specified in text files. The SPE allows individual programs to be coded without knowledge of other parts of the system and thus allows systems to be quickly built, modified, or scaled without program recompilation.

The SPE has been implemented and tested on an Intel Paragon XP/S 25. The SPE continues to evolve as it is used with additional applications. Although the current implementation interfaces to the operating system using Intel-specific NX calls, it should be portable to the emerging Message Passing Interface standard or to other vendor-specific parallel operating system interfaces based on message passing.

The SPE has been successfully used by several acoustic and SAR processing projects. Use of the SPE has provided more rapid program development and system integration, and has resulted in applications that are scalable and reconfigurable.

CONTENTS

Appendices

Figures

1. INTRODUCTION

The Scalable Programming Environment (SPE) is a programming environment and system interface that was developed by the Hybrid Digital Optical Processor (HyDOP) and the Bottom Limited Active Classification (BLAC) projects, both sponsored by the Office of Naval Research (ONR-321), to help build large scalable real-time systems in a research and testbed environment. It provides the user with the ability to build and modify scalable systems quickly using both function- and data-domain decomposition methods.

The SPE allows a programmer to easily write scalable applications. It loads and runs heterogeneous programs on multiple sets of nodes and provides the scalable data-path connections needed for unrelated parallel programs to communicate.

Each program in a *system*, or an *application*, executes on a set of nodes and performs a different function. Each node for a given program executes the same code, called an *instance* of the program. Each instance of a program is expected to work on a different piece of the data for the given program. The SPE provides the complex high-level message-passing routines that are needed to interconnect different programs and instances of programs into a system.

1.1 THE NEED FOR APPLICATION SCALABILITY

In a research and development environment, where change is a fact of life, application scalability is fundamentally important to successful project implementation. Scalability allows the application to be placed on the hardware in such a way that optimum use is made of each of the hardware resources: compute cycles, memory, and communications bandwidth. Even when running in real time is not an issue, it is still desirable to balance the resource usage for each program configuration so that overall run time is minimized.

Determining resource usage during the initial analysis and design phases of a project is a very difficult problem, and the variability inherent in a testbed environment assures that all but the most general analysis will instantly be out of date. There are two sources of this variability: the project itself and the underlying computer system.

Project changes have a number of sources. Most fundamentally, the project requirements change and evolve: customers change their minds, funding levels are reevaluated, and new technologies are discovered. Researchers may wish to experiment with new algorithms, each of which has different resource requirements. In a signal processing environment, developers may add new functions to the signal processing path or remove others, and those resources must be taken from, or made available to, other parts of the system. Some investigations may focus on particular algorithms, so the analyst may devote a high proportion of the resources to a given algorithm at the expense of others, for instance, to improve the processing fidelity for a specific data set. For many reasons, including some just mentioned, the system will be assembled in various configurations such as real-time, non-real-time, and development configurations, and each configuration requires a unique allocation of resources.

The supporting computing system may also change in ways that require the application resources to be allocated differently. A new release of the operating system may use more memory, leaving less for application programs and requiring more nodes to be used for the same processing. Compiler modifications may improve (or degrade) the computing efficiency of a node for a particular algorithm. System software changes may improve the communications bandwidth. And, of course, the

underlying hardware may be upgraded to increase memory size, communications bandwidth, or processing power.

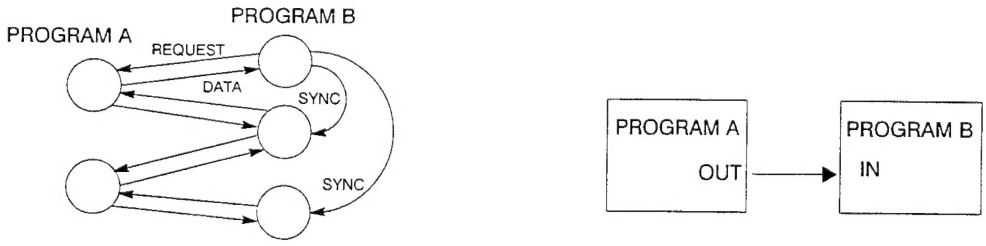
Given the cost of large parallel computers, it is likely that project development will be done on a computer that is being shared with other users. Under these circumstances, machine resources may have to be allocated on a run-by-run basis depending on the availability and usage of the system.

For all of these reasons, constant fixed resource allocations are impractical in this environment. A system must be provided to allow users to transparently reallocate resources to the various functions composing their applications.

1.2 CURRENT PRACTICE

Currently the parallel computing industry does not provide a standard set of high-level message-passing routines to systematically interconnect multi-instanced heterogeneous programs in a system. These systems must be built with the details of the message passing visible to the application programmer. A user developing a multi-instanced program must know the intricacies of the other interconnected programs, how the data are shaped on the other end of the communications paths, how the program will control the flow of the data it receives, how it will buffer and transform the data once received, and how it will present the data synchronously to multiple instances of itself.

Figure 1a shows the level of message-passing detail that a traditionally developed multi-instanced program must contend with in a heterogeneous system. Each instance of program B must contain explicit code to correctly handle the various communication messages. The code in each instance is dependent on how the data are shaped at the other end of each communications path. Each instance will contain code to explicitly control the flow of data it receives (request messages). If program B receives messages from two or more programs (not shown), then the program must guarantee that each instance receives all messages in the same order as all other instances (controlled by the synchronizing messages).



(a) Traditional view. (b) SPE view.

Figure 1. Message passing between heterogeneous programs.

This level of message-passing detail is beyond the level that an application programmer should have to worry about. Furthermore, it is time consuming and prohibits prototyping large systems or modifying existing ones. Involving the user at this level of detail has hindered the emergence of system designs beyond the multi-instanced single-program standard predominantly used today.

1.3 SPE FEATURES

The SPE has been developed to hide this level of message-passing detail. Figure 1b shows the view that an SPE program sees when communicating data. Communication between programs is via *ports* connected by *nets*. Therefore, within each program, there is no requirement for information

about which programs are connected, the number of program instances for this program, the number of program instances at the other end of each communication path, how the data are buffered, how the flow of data is controlled, or how the data are synchronized.

Furthermore, because this level of detail is hidden from the program, new systems can be quickly built and old ones quickly modified. Programs of a system can be scaled to run on any number of processing nodes while requiring no changes to source code. Programs can be disconnected and reconnected in different ways to modify the function of a system.

The SPE message-passing routines have been built to utilize resources efficiently. They have been designed to overlap processing with communication, minimize buffer space, avoid extra copying of data, and minimize the number of messages.

The SPE includes other features useful in a parallel programming environment, such as run-time control of diagnostic flags and execution parameters, data capture, performance monitoring, logging, and error reporting.

The SPE provides:

- A loader, which allows the user to define, load and run parallel programs on scalable sets of nodes without the need to recompile. New systems can be built or modified by changing a *System Definition* file. Systems can be easily run on varying numbers of nodes to change system performance or to meet constraints of the hardware system.
- High-level message-passing routines to transfer data between programs running on differing numbers of nodes. Multiple programs are interconnected with data-flow-type connections, which hide the parallelism of the system within the connections. The message-passing routines provide the scatter-gather operations needed to pass data between programs running on differing numbers of nodes, provide internal synchronization controls to make messages received by programs synchronous to every instance of a program, and provide first in, first out (FIFO) data buffers so that programs sending and receiving messages between each other can work on different-size data blocks.
- A debugging environment for data capture, performance monitoring, logging, and error reporting. Debugging a parallel application requires a user interface that deals with multiple programs and multiple instances of programs. The SPE provides *spe_report()*, a *printf()*-like call, which conditionally writes to standard output based on a run-time parameter which can be unique to each *spe_report()* call. Other routines allow the user to record performance statistics and to view a summary at the end of the run. The data being transferred between any two programs may be written to an external file in several formats; the SPE handles gathering together the distributed data.
- A global database for the storage of symbolic names with their associated values. Parameter values within a system can be set at run time so that application code need not be recompiled when values are changed. When new names are added, programs that don't use them do not need to be recompiled.

1.4 SPE STATUS

Development of the SPE is continuing since, when it is used with new applications, new requirements are often identified. The current implementation runs on an Intel Paragon XP/25. Although the current implementation interfaces to the operating system using Intel-specific NX calls, it should

be portable to the emerging Message Passing Interface (MPI) standard or to other vendor-specific parallel operating system interfaces based on message passing.

1.5 ORGANIZATION OF THIS REPORT

This report is organized as follows:

Section 2 describes how parallel programs communicate through ports. Different types of ports are described that define how data are scattered and gathered when communicated between parallel programs. Also discussed is how data are buffered and synchronized between programs.

Section 3 describes the user interface. This section also describes the input text files through which the user describes an SPE application. These files define each SPE program, describe how they are connected to form a system, and initialize database variables that can be used by the programs. Examples are shown for each of the files, and rules are provided describing the semantics.

Section 4 describes the programming interface. The various SPE routines are described and examples are provided showing how they are used in a typical application.

Section 5 shows how to compile and run an SPE application. The calling arguments and options of the SPE program itself are described, and details are given on the interactive runtime interface.

Appendix A summarizes predefined database variables that can be used by the user to obtain diagnostic information from the SPE. Appendix B lists the reserved words recognized by the SPE when interpreting the input text files. Appendix C defines each of the SPE programming calls. Appendix D describes the format and grammar used for the various description files. Appendix E shows the decomposition algorithms used by the SPE when gathering or scattering data for a port across multiple instances of a program.

1.6 ACKNOWLEDGEMENTS

1.6.1 Sponsors

The original development of the SPE was supported by the HyDOP and the BLAC projects, sponsored by the ONR-321.

Incentives to implement some features of the SPE were provided by the use of the SPE to implement synthetic aperture radar (SAR) and interferometric SAR (IFSAR) image formation and automatic target recognition (ATR) applications. This work was supported by the Advanced Research Projects Agency (ARPA) Computing Systems Technology Office (CSTO) and Advanced Systems Technology Office (ASTO).

The SPE was developed using a Paragon machine provided by the Department of Defense High Performance Computing (HPC) Modernization Program Office.

1.6.2 Software Tools

The work reported here was helped appreciably by the following freely available software tools:

- The Revision Control System (RCS) written by Walter Tichy and distributed under the GNU license. An imbedded version of the RCS *indent* command is used for SPE version consistency checking and application program version logging.

- *cpp* from *gcc*, the GNU C compiler. An embedded version of *cpp* is used to preprocess all SPE description files.
- DDB, a package of dynamic memory database routines written and made available by Christopher G. Phillips.

2. COMMUNICATION IN THE SPE

The SPE distinguishes between communication of *data-flow* and *control* information and treats the two in different ways. A data-flow represents a continuous stream of information from one set of nodes to another set in which the format and rate of the data remain constant during execution of the system. Data-flow information also has inherent parallelism which the SPE exploits by distributing parts of the data to different instances of a program. Control information, on the other hand, does not flow at a constant rate, may not always be produced, and the size or structure of the information may vary during the course of execution.

All communication from one program to another within an SPE application is done through *ports*. This allows a program to be written in such a way that it is independent of its interconnections, thereby allowing the user to rearrange program configurations at the time the program is loaded. The SPE allows ports to be specified as *striped*, *replicated*, or *control* ports.

2.1 DATA-FLOW PORTS

Specifying that a port is a *striped* or *replicated* port tells the SPE how it should decompose a data-flow connection as the data are transferred between ports of programs of multiple instances. When the SPE transfers data to an input port that is replicated, then all instances of the receiving program will be given the same data. When the SPE transfers data from an output port that is replicated, then each instance of the sending program must provide the same data. When the SPE transfers data to an input port that is striped, then each instance will be given a different (possibly overlapping) portion of the data. When the SPE transfers data from an output port that is striped, then each instance of the sending program will provide a different portion of the data (cannot be overlapping). The SPE can transfer data between ports of similar or dissimilar types.

2.1.1 Striping Data

The size of the data communicated over a striped or replicated port must be defined by the user in two dimensions, rows and columns. The data do not actually have to be two-dimensional, but to the SPE it must be described as such (i.e., [1][1], [1][5], and [5][1] are valid). When the SPE transfers data to a striped port of a program of multiple instances, it divides the data along its row dimension. The data are not decomposed across the column dimension. The algorithm divides the data as equally as possible into row-contiguous portions (see Appendix E). Figure 2 shows how a 6×4 array of data would be striped across programs of 3 and 2 instances.

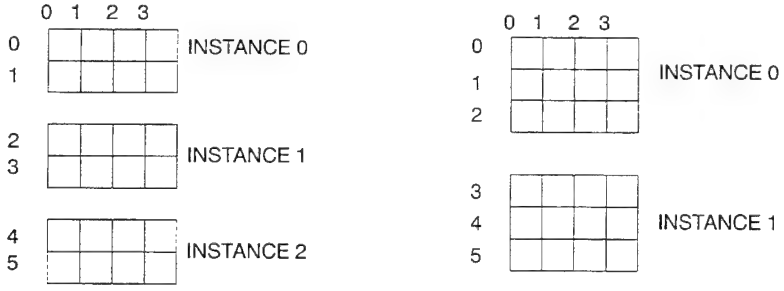


Figure 2. Striped 6-row by 4-column array.

When communicating data between programs, the SPE needs information on how the data are decomposed at the sending and receiving ends of a connection. For a simple two-port connection, the

SPE must be able to handle eight basic types of connections: striped-striped, striped-replicated, replicated-striped, replicated-replicated, striped-transposed-striped, striped-transposed-replicated, replicated-transposed-striped, and replicated-transposed-replicated. Figure 3 shows what the communication paths might be for each type of connection. The examples show the sending program running on 3 nodes and the receiving program running on 2 nodes. The data that are communicated are in a 6-row by 4-column array. The extent of the data communicated on each path is shown as [*rows*][*columns*].

Note that for the cases in figure 3 where the sender is replicated, there are many possible implementations, only one of which is shown. Also note that two pairs of diagrams (b and f, d and h) appear identical but are differentiated by the fact that the received data are transposed before being copied into the receiving buffer.

One can see that even for these simple cases, the level of detail is quite complex. Information must be provided to the SPE about where each program instance is sending or getting its data and how it will scatter or gather its data. Each instance within a program will operate differently to communicate its portion of the data. Also, if the sending or receiving programs are scaled to run on a different number of nodes, or if additional ports are added to the net, then the paths will change.

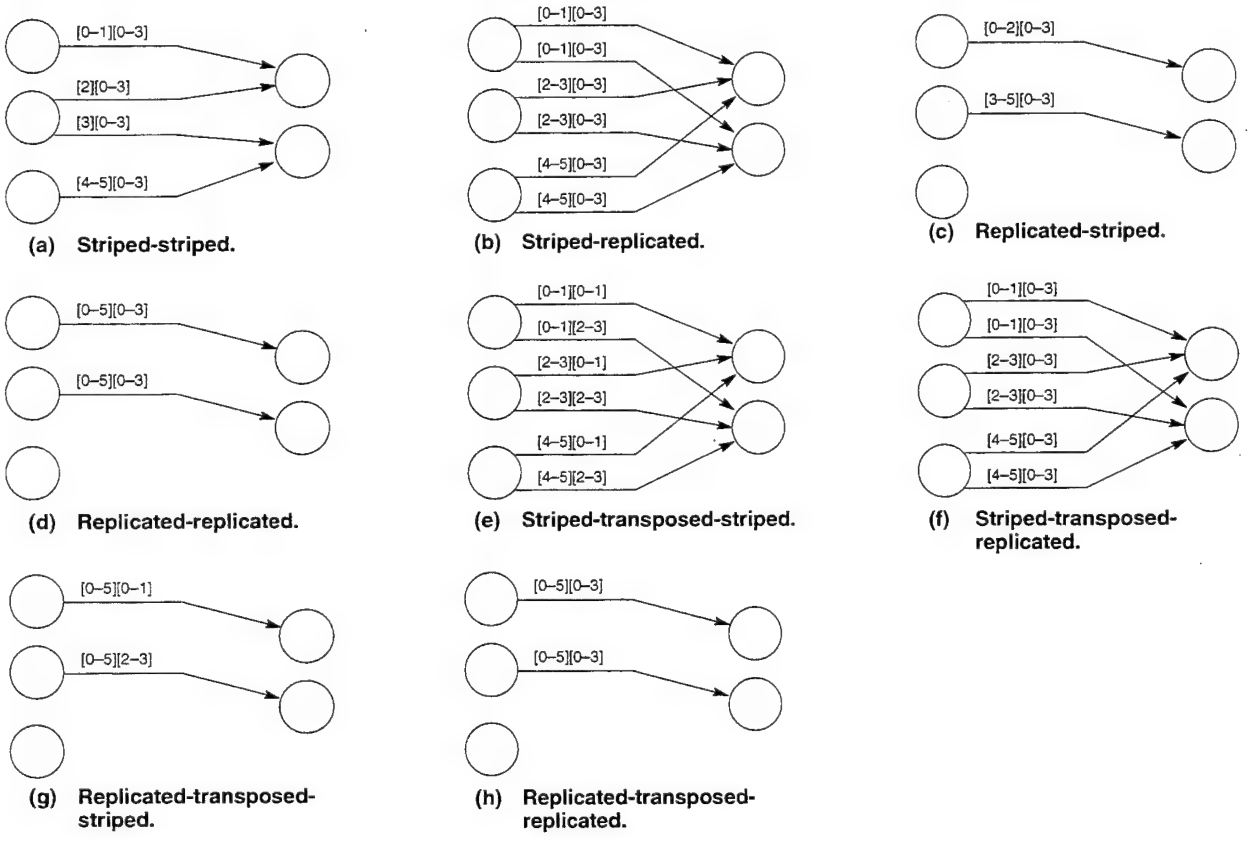


Figure 3. Internal message paths for communicating a 6-row by 4-column array.

An application programmer should not have to deal with this level of detail, and indeed this is something that is provided for and kept hidden by the SPE. The programmer should not be concerned with where the program is getting or sending its data or how the data are decomposed at the other end of a communication path. Nor should the code be dependent on executing as a particular

instance of the program. The only thing the programmer should need to know is what portion of the data the program works on and must produce.

2.1.2 Overlapping Striped Data

In order to take advantage of striping data to multiple program instances, the application code must be able to process each data row independently of other rows. This allows processing to proceed with no further communication between program instances. Some algorithms, however, require the availability of the data in a number of adjacent rows. The SPE addresses this requirement by allowing the user to specify an *overlap* for striped ports. This overlap is given by the number of rows of overlap needed, and can be symmetrical or asymmetrical. That is, considering the block of rows allocated to a particular program instance, the number of rows of overlap at the beginning of a block can be the same (symmetrical) or different (asymmetrical) than the rows of overlap at the end of the block. In addition, the SPE allows the overlap on the first and last instances of a program to be treated differently than on the other instances, a feature required by some algorithms.

2.2 CONTROL PORTS

Ports can be specified to the SPE as being of *control* type. Control ports are provided to allow programs to communicate data that are not a part of the normal data-flow of the system. Data sent over a control port do not have specifications for array size or element size, and the stripe and block overlap options do not apply. When the composition of data sent between programs is irregular (cannot be specified as a two-dimensional matrix), unknown, or varies during the run, then the data must be sent via control ports. Control ports can only be connected to other control ports.

Normal control ports are connected like the replicated-replicated connection shown in figure 3d (ignoring the dimensions of the data). That is, the same data must be sent by each instance of the sending program and each instance of the receiving program gets the same data. However, *sequential* control ports have different communication patterns. A sequential output control port creates a virtual data flow consisting of an ordered sequence of control messages. Each program instance writes messages whenever appropriate for the data subset being worked on by that instance regardless of whether other instances write messages at the same time. Note that this is considerably different than data-flow and non-sequential control ports where every instance must make matching send and receive port calls. The order in which messages are transmitted through a control sequence port is the order in which the instances make their request to send data. The SPE guarantees that all receiving programs get the messages in the same sequence.

When data from a sequential output port are sent to an input port, each message is normally delivered to every instance of the program. However, a *round-robin* input control port routes each subsequent message to a different instance of the program. When the time to process each incoming message is the same, this provides a transparent method of parallelizing the processing of a sequence of messages.

Figure 4 below shows an output control sequence port connected to an input round-robin port. The three instances of the output port write messages whenever they wish. In the example shown in figure 4a, instance *a* writes two messages, then instance *c* writes one message, while instance *b* sends no messages at all. These messages appear to the user as the *virtual* sequence of messages shown in figure 4b. Finally, figure 4c shows the message stream being distributed to the two instances of the destination program in round-robin order. This model of a virtual message sequence simplifies the user's view of the interconnections; in fact, however, the SPE implementation delivers each message directly from source instance to destination instance as efficiently as possible.

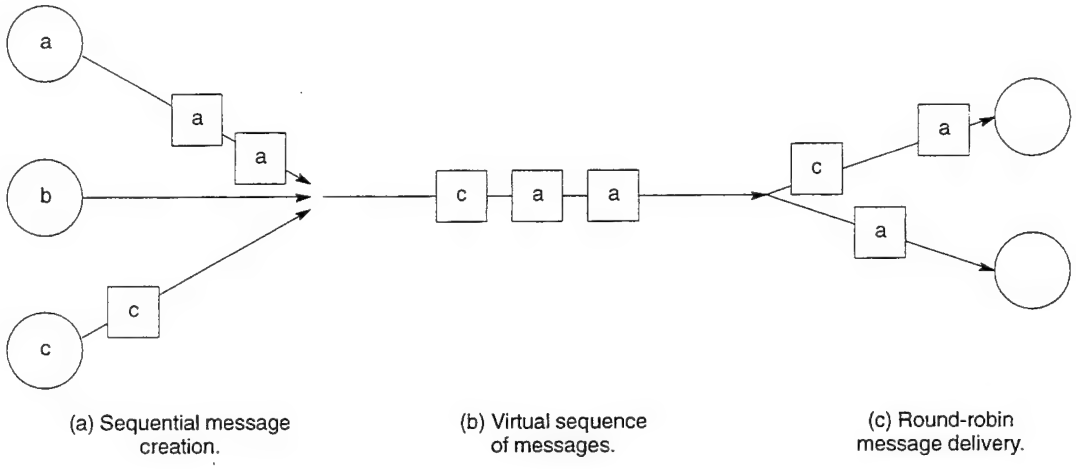


Figure 4. Sequence port to round-robin port connection.

2.3 FIFO BUFFERS

Another feature the SPE provides for data-flow input ports is that two interconnected programs can work on different-sized data blocks. This supports the concept that a data-flow connection is a stream of data columns flowing from one program to the next. The SPE requires that the *row* dimension of each input port on a net agree with the output port to which it connects, but allows the *column* dimension of each to be different. The SPE can allow this by providing a FIFO buffer on each input port which stores the data when the data are received. The FIFO buffer is a two-dimensional buffer that performs the FIFO operation along the columns dimension of the buffer. Since the communicated data is two-dimensional, the FIFO buffer must also be two dimensional. Figure 5 shows the operation performed by the FIFO buffer.

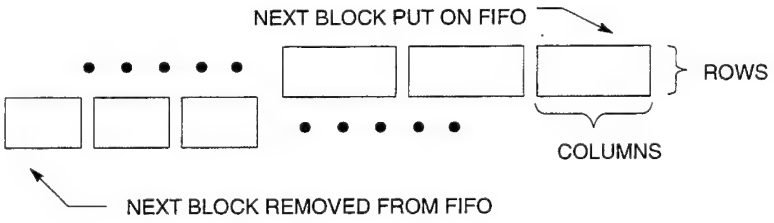


Figure 5. Input port two-dimensional FIFO.

The implementation of two-dimensional FIFO buffers is complicated by the fact that the physical memory of a computer is accessible in only one dimension (every memory location is accessed by one address). To understand this point, consider figure 6.

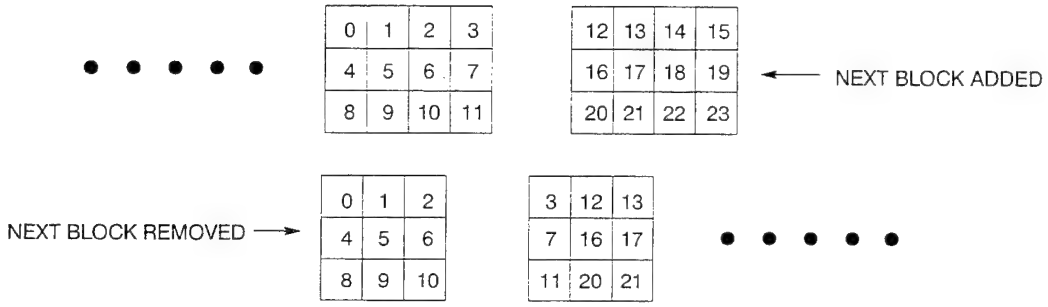


Figure 6. Memory addresses in two-dimensional FIFO.

The 3 by 4 element boxes at the top of figure 6 represent the data being put into the FIFO buffer. The 3 by 3 element boxes at the bottom of the figure represent the data being removed from the FIFO buffer. (Looking back at figure 3a, we see this represents the data sent along the top two paths.) The number in each box represents the address where each element is stored. One can see that if the elements of the input FIFO blocks are stored contiguously in memory, then the elements of the output FIFO blocks will have to be read from noncontiguous memory locations. For instance, the first block removed is read from addresses 0, 1, 2, 4, 5, 6, 8, 9, and 10. Once again, this level of detail is beyond what an application program should be concerned with and is a feature provided by the SPE.

The SPE also supports *block overlap,* which allows the blocks of data taken from a FIFO buffer with one receive call to overlap the data received with the next. The amount of overlap (in data columns) is specified for each input port in the Program Definition files.

Another important benefit that a FIFO buffer provides is that it allows a program to overlap communication with computation. For example, when a program is working on a block of input data, it can also be receiving in its input FIFO buffer future blocks of data. Thus, when it finishes working on the current block of data, it is ready to start to work on the next. The SPE provides the internal control signals sent between the receiving and sending programs to keep the FIFO buffers full.

2.4 MESSAGE FLOW CONTROL

When a sending program sends data to an output port, the program blocks until the SPE has sent the data to all the input ports the output port is connected to. The SPE will send the data to each input port as space is made available in the port's input FIFO buffer. When all the input ports have received the data in their FIFO buffer, the SPE returns control to the sending program.

When a receiving program receives data from an input port, the program blocks until the data become available in the port's input FIFO buffer. When enough data have been collected in the FIFO buffer to satisfy the input request, the SPE will transfer the data to the user's buffer and return control to the receiving program.

Internally the SPE controls the flow of data by having the receiving program tell the sending program when it can send more data. Just before the SPE returns control to the receiving program, it makes a decision of whether or not to let the sending program send more data. If the receiving program has room in its FIFO buffer for more data from the sending program, it tells the sending program how much more data to send. While the data are being sent, the SPE returns control to the receiving program, allowing it to process the current buffer of data. The cycle repeats itself each time the receiving program receives data.

Although control ports have no FIFO buffers, these ports otherwise work as described above.

This method of flow control provides the programmer with a simple decentralized method for synchronizing programs. No program has to contain information about the requirements of the programs it is connected to nor does any have to generate control signals to control the flow of data it consumes or produces. Instead, these control signals are handled internally by the SPE. Each program simply receives and produces data as fast as possible.

2.5 MESSAGE ORDER

There are other issues concerning data communication besides how the data are connected or buffered. In parallel processing, where multiple instances of a program work on a problem, the data flow seen by each instance of the program must be coherent; that is, messages received by each instance of a program must be received in the same order. This is an obvious requirement for the correct operation of a program that receives data from multiple programs which operate asynchronous to each other.

The SPE satisfies this requirement by providing several guarantees. First, the SPE guarantees that messages from a given node are received in the order they were sent. Then the SPE makes sure that messages received first are always delivered first. Finally, the SPE guarantees that for programs having multiple input ports, the messages will be received in the same order by each instance of the program.

Figure 7 shows the control signals used by the SPE to provide the message-passing synchronization needed between program instances and to buffer the data between sending and receiving programs. The *request* lines from program B to program A indicate that the FIFO buffers in B are ready for more data and also indicate how empty they are. The *sync* lines from B0 to B1 and B2 indicate the order in which B0 has received its messages. The SPE uses this information to force messages to be received by the user's program on B1 and B2 in the same order as on B0.

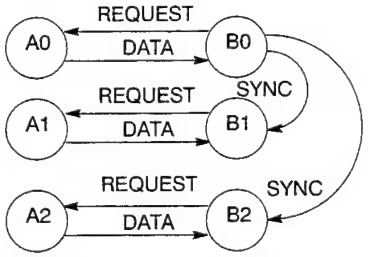


Figure 7. Control signals.

3. USER INTERFACE

The user controls the SPE through a *System Definition* file, *Program Definition* files, and *Database Startup* files. The System Definition file and Program Definition files describe in two levels how each program in a system interfaces to other programs in the system, and how each program interfaces to its outside world. The Database Startup files are used by the SPE to enter variables into a global database that can be used by each program in the system.

A user loads and runs an SPE application by executing the program `spe` and supplying it with a path name to the System Definition file for the application. When `spe` starts, it reads the System Definition file and Program Definition files. From these files, it determines the configuration of the system, and loads the programs specified in the System Definition file onto nodes of the target hardware. `spe` then creates and downloads a unique *port map* to each program instance in the system, which describes exactly how each program is connected to the other programs of the system. `spe` then reads the Database Startup files as specified on the command line and initializes the SPE database. Once all the programs have received their unique port map and the SPE database is initialized, the programs are ready to run. `spe` interacts with the user through standard input and output.

The `spe` has a number of features that provide flexibility for the input files. Comments (prepended with two slash characters "//") allow documentation of the files and increased readability. All input files are filtered through a C preprocessor (GNU `cpp`) that allows for file inclusion and macro expansion as specified by the C language. `spe` also does expression evaluation for further flexibility and scalability of the input files.

Details of the format and grammar of these files are described in Appendix D. See Section 5.3 for a complete description of all the arguments that may be provided to the `spe` program.

The content of the input files, after preprocessing, must conform to the file formats described in the following sections. In each of these sections, an example input file is given that illustrates a possible description file for the example system shown in figure 8. Each box represents a different program in the system, with a number in the lower right-hand corner indicating the number of nodes on which it runs. Each box has named ports through which it communicates data to other ports on its net. Input ports that are additionally buffered or transposed have an attached bubble containing an integer or the letter "T," respectively.

3.1 SYSTEM DEFINITION FILE

The System Definition file defines which programs are used in a system, how many nodes each program will run on, and how programs are interconnected. `spe` uses it to determine what programs will be loaded on what nodes and to make a unique port map for each program instance. Figure 9 is an example of a System Definition file.

The System Definition file specifies the path name to each program that is to be loaded by `spe`, the number of nodes that each program will run on, and a path name to the Program Definition file for each program that will be loaded. After reading the System Definition file, `spe` then reads each Program Definition file so that it can completely determine the port map for each program in the system. The port map describes for each program instance what portion of the problem it will work on and how it is connected to other programs in the system. `spe` determines which portion of the data each program instance will work on based on the Program Definition file for a program and on the number of nodes on which the program is specified to run.

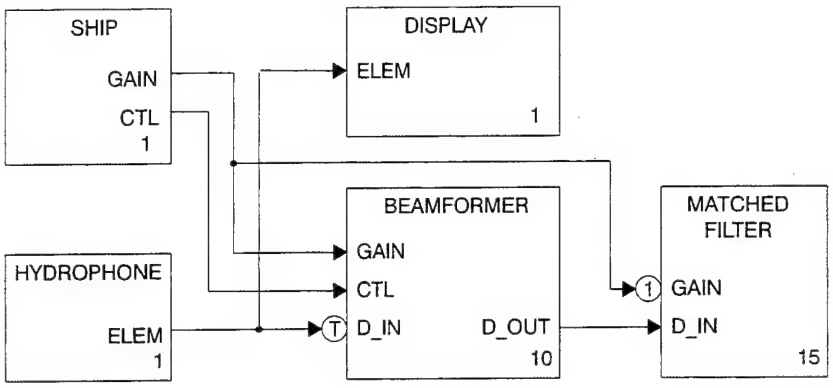


Figure 8. Example showing an implementation of an acoustic receiver system.

```
// File: system/receiver
//
// Key Word   #Nodes         Program          Prog_Def_Path          Executable_Path
// ---------------------------------------------------------------------------------
   PROGRAM    1              ship             "def/ship"             "bin/ship"
   PROGRAM    1              hydrophone       "def/hydrophone"       "bin/hydrophone"
   PROGRAM    1              display          "def/display"          "bin/display"
   PROGRAM    (10,20,0.5)    beamformer       "def/beamformer"       "bin/beamformer"
   PROGRAM    (15,40,1.0)    matched_filter   "def/matched_filter"   "bin/matched_filter"

// Key Word   Port
// ---------------------------------------------------------------------------
   BUFFER     matched_filter:gain    1
   TRANSPOSE beamformer:d_in

// Key Word   Net_list
// ---------------------------------------------------------------------------
   NET        ship:gain, matched_filter:gain, beamformer:gain
   NET        ship:ctl, beamformer:ctl
   NET        hydrophone:elem, beamformer:d_in, display:elem
   NET        beamformer:d_out, matched_filter:d_in

// dump all input data to the beamformer (after transposition)
   DUMP       beamformer:d_in [:][:] SPE="input data"
// dump rows 10 through 20 of frame 5 in MATLAB format
   DUMP       beamformer:d_out [10:20][:] MATLAB="float_complex" FRAMES=5
```

Figure 9. System Definition file corresponding to the system shown in figure 8.

The System Definition file also specifies the *net lists* that interconnect the ports of each program in the system. Programs communicate by passing input and output data through their ports. Data written to an output port of a program are sent to the input ports on the net list to which the output port is connected. A program is unaware of where its data go or where the data come from. This basic principle of the SPE allows users to build modular systems that can be quickly modified or extended to meet the changing needs of an application, and also promotes the reuse of software when new applications are built.

The System Definition file specifies whether the data received by an input port need to be *transposed* or *additionally buffered* to meet the input requirements of a program. Data must be transposed when two connected programs need to process their data along different dimensions (such as in a

two-dimensional fast Fourier transform (FFT)). Additional buffering on an input port may be needed to improve efficiency by increasing the amount of data the port can store.

Within the System Definition file, the user may specify that data sent or received on a striped or replicated port also be written (dumped) to a file. The SPE handles gathering the distributed data into a single file record. In addition, the user may specify that a subset of the transmitted data be recorded that can often result in an important reduction in file size. Data can be dumped in MATLAB format, in ASCII format, or in a binary format unique to the SPE. Data frames dumped from ports on programs that have multiple instances are written to the file as one logical record. Data dumped from different ports of the same program are normally written as multiple records to the same file, although this can be changed with the use of options. On each invocation of the SPE, the first write to the file normally erases the previous contents of the file; this behavior can also be controlled with an option. Each record written to the dump files contains a header section that names and describes the data in the record. The name given to a data record is composed of the port name and the frame number of the data. For `MATLAB` files, the name of the record becomes the variable name. By use of the optional dumping controls and multiple `DUMP` commands, multiple ports may be dumped to the same file and different subranges of data from a single port may be dumped to one or more files.

The rules for making a System Definition File are as follows:

1. Each line of the System Definition file must begin (ignoring white spaces) with `PROGRAM`, `BUFFER`, `TRANSPOSE`, `EXCLUDE`, `NET` or `DUMP`. These reserved words determine the format of the rest of the line.
2. Each line starting with `PROGRAM` specifies a program used in the system. The first field is a specification of the number of instances of the program (i.e., the number nodes on which the program runs). This can be given as a single integer, or as three values `(min,max,weight)` that give the minimum and maximum number of nodes and a node allocation *weighting factor*. The weighting factor is used in conjunction with the weighting factors of the other programs to allocate any available nodes after the minimum requirements are met. When allocating the additional nodes, the SPE attempts to match the ratio of the total number of nodes allocated to each program to the ratios of the weighting factors. For instance, in the example of figure 9, the first five additional nodes are allocated to the matched filter program giving it a total of 20 nodes. Since the beamformer program has the minimum 10 nodes, this brings the total number of nodes allocated to 20 and 10: the 2-to-1 ratio of the weighting factors. After this, two additional nodes will be allocated to the matched filter program for each one allocated to the beamformer program.

   The second field is an identifier specifying the symbolic name by which the program will be referenced in other parts of this file and the Database Startup files. The third field is a string that specifies the path name to the Program Definition file. The fourth field is a string that specifies the path name to the executable that runs on the nodes. The path name field can include program arguments that are separated by spaces within the string.
3. Each line starting with `BUFFER` specifies an input port that needs extra buffering to store data received on that port. The amount of additional buffering requested is expressed in integer multiples of the normal buffer size provided by the SPE. The normal buffer size is dependent on a number of factors internal to the SPE.
4. Each line starting with `TRANSPOSE` specifies an input port that must have its data transposed when received.

5. Each line starting with EXCLUDE specifies a program that, although described elsewhere in the System Definition file, is ignored by the SPE. This makes it easy for the user to work with different configurations of the same program without continually modifying other parts of the System Definition file.

6. Each line starting with NET specifies a list of ports that are connected together in a net. Ports are formatted as *program:port*, where *program* and *port* are replaced by identifiers. The first port in a net must be an output port. The remaining ports must be input ports. Ports do not have to be connected to a net. (There is an SPE call that can check if a port is connected, *spe_port_is_connected()*.)

   Control ports can only be connected to Control ports (see Section 2.2).

   The *rows* dimension of all nontransposed input ports on a net must agree with the output port to which it connects. The *rows* dimension of all transposed input ports on a net must agree with the *columns* dimension of the output port to which it connects.

7. Each line starting with DUMP specifies a port whose data are to be dumped to a file. The first field specifies the port to be dumped. Ports are formatted as *program:port*, where *program* and *port* are replaced by identifiers. Data can only be dumped from striped or replicated ports. A port that is not connected can nevertheless dump data; this allows specification of unused ports for the purpose of viewing internal data.

   The second field indicates the rows and columns of the data array to be dumped. Rows and columns are specified as *[first_row:last_row] [first_col :last_col]* indicating the range (inclusive) of rows and columns to dump. Any or all values may be left empty indicating that the first or last row or column should be dumped. See figure 9 for examples.

   The third field is specified as *filetype=format*. *Filetype* specifies the type of the output data file and must be one of MATLAB, ASCII, or SPE. The *format* field is a string that gives the type of data on the port. If *filetype* is MATLAB or ASCII, the format string must be one of:

```
"double"    "double_complex"
"float"     "float_complex"
"int"       "int_complex"
"short"     "short_complex"
"ushort"    "ushort_complex"
"uchar"     "uchar_complex"
```

   When *filetype* is SPE, the format may be an arbitrary string that will be included in the record header.

   A number of options may be given in any order at the end of the DUMP specification line:

   - APPEND — The output file will not be erased when the first port record is written. Rather, new records will be appended to the file.
   - FILENAME="*string*" — The output file name is changed to "string". By default, the file is named "*programname.extension*" where *extension* is one of "mat", "ascii", or "spe" depending upon the specified *filetype*.
   - FRAMES= *start:stop* — This option specifies which data frames to dump. *Start* and *stop* indicate the range of frames to dump. Frame numbers start with 1 and increase. To dump a single frame, the colon and *stop* frame indicator are omitted. By default, all frames are dumped.

- LABEL="*string*" — For an SPE filetype, an arbitrary string may be specified for inclusion in the header. This allows the user to include with the data a description of its source or processing history.
- NO_HEADER — For an SPE filetype, the header information may be omitted in the case that the user needs a file containing just the raw data. Normally, a header is included that describes the data in the dump file. For the exact description of the data header, refer to the SPE_FILE_HEADER type definition in "spe.h" (see Section 5.1).
- RENAME="*string*" — The output port record is renamed to "string_*xx*" where *xx* is the frame number. By default, every record is given the name of its port and frame number.
- STRUCTURE= *offset,count,stride* — If the array element is a data structure, this option allows the amount of data dumped to be limited to a portion of the structure. *Offset* is an offset in bytes into the structure indicating where a subarray of data is located. The number of elements of data to dump is specified with *count*. If the *count* field is given as ALL, then the number of elements dumped is derived from the data block size, the given starting offset, and the stride. The optional *stride* argument specifies the distance in bytes between elements. If *stride* is omitted, it is derived from the output format. The type of the output data items is that indicated by *format* in the *filetype* field.
- CONVERT=*convert_type* — Normally, the SPE checks that the type of the output data items specified by *format* in the *filetype* field is consistent with the port element size. When the user is converting port data elements before dumping, the sizes of the types before and after conversion may differ. This checking is disabled when the CONVERT option is used. The optional *convert_type* field is an integer that is passed to the user-specified dump conversion routine (see the entry for *spe_dump_define()* in Appendix C).

3.2 PROGRAM DEFINITION FILE

The Program Definition files define each program's input and output. They are used, along with the System Definition file, to make a unique port map for each program and instance in the system. Each Program Definition file defines only the input and output for its own program, that is, there is no information in it defining what the program is connected to. For each use of a program in a system, there can be a different Program Definition file.

A Program Definition file defines each port of a program. A port is defined by its *direction*, *type*, *array size*, *element size*, and other options specific to certain types of ports.

The port *direction* indicates whether a port is an input or output port. Other fields of a port's definition can or cannot be specified, depending on its direction.

The port *type* describes whether or how data will be decomposed among a program's instances when data are received or sent from a program. The port type can be defined as *striped*, *replicated*, or *control* (see Section 2 for descriptions of each).

The port *array size* defines the size of the data a port receives or sends. It is specified only for replicated and striped port types. If specified, it defines a port by two dimensions, rows and columns. It need not actually be two-dimensional, but to the SPE it must be described as such (i.e., `[1][1]`, `[1][5]`, and `[5][1]` are valid).

The port array size defines the size of the data before decomposition. That is, if the port is striped, then each instance of a program will see only its portion of the data. If the port is replicated, then each

instance will see all the data. For striped ports, the data are decomposed across rows of the array (see Section 2.1).

If a port is replicated, then the rows dimension can be any value. If it is striped, then the number of rows must be greater than or equal to the number of program instances.

The port *element size* defines the size of each element of the data when the port array size is specified. The port element size will vary depending on the data that are processed (i.e., complex, real, etc.).

The port *stripe overlap* option defines how many rows of overlap to use when decomposing data across a striped input port. For striped ports, the SPE decomposes the data across rows of the array. When the data are overlapped, adjacent program instances share common rows of the data between them. The port stripe overlap can be specified only for input striped ports. Appendix E specifies the algorithms used for decomposing striped overlapped and nonoverlapped data over program instances.

The port *block overlap* option allows columns of a data block taken from the input buffer to overlap with the next receive. This option is explained in Section 2.3 where the input FIFO buffers are described.

Output control ports may be specified as *sequential*; input control ports may be specified as *round-robin*. See Section 2.2 for a description of these port types.

Shown in figure 10 are examples of Program Definition files that could have been used in the receiver system example in figure 8.

The rules for making a Program Definition file are as follows:

1. Reserved words that can be used in Program Definition files include `PORT`, `INPUT`, `OUTPUT`, `STRIPED`, `REPLICATED`, `STRIPED_OVLP`, `BLOCK_OVLP`, `CONTROL`, `SEQUENCE`, and `ROUND_ROBIN`.
2. Each line starting with `PORT` specifies the definition of a port. The fields are *port, direction, type, array size, element size,* and options.
3. The *port* field is an identifier specifying the name by which the port will be referenced.
4. The *direction* field must be specified as `INPUT` or `OUTPUT`.
5. The *type* field must be specified as `CONTROL`, `STRIPED`, or `REPLICATED`.
6. The *array size* and *element size* fields must be and can only be specified for striped and replicated ports. The format for the *array size* field when specified is *`[rows][columns]`*, where *`rows`* and *`columns`* are integers. The *element size* field is the number of bytes for each element. On input ports, any of these three fields may be given as the reserved word `ANY` in which case the corresponding value is derived from the output port to which this input port is connected.
7. There are several optional fields that may be specified at the end of a `PORT` definition line. These are specific to certain kinds of ports:
   - `BLOCK_OVLP=`*overlap_size* — The *block overlap* option can only be specified for input ports that are striped or replicated. The value of *overlap_size* must be an integer less than the number of columns of the input data. Block overlap is described in Section 2.3.
   - `ROUND_ROBIN` — The *round robin* option can only be specified for `INPUT` control ports. See Section 2.2 for a description of round-robin control ports.

- SEQUENCE — The *sequence* option can only be specified for OUTPUT control ports. See Section 2.2 for a description of sequenced control ports.
- STRIPED_OVLP=*overlap_spec* — The *striped overlap* option can only be specified for input ports that are striped. The *overlap_spec* argument specifies one of several overlap options as well as the size of the overlap. Section 2.1.2 describes how data overlap works in general, while the decomposition algorithms are described in detail in Appendix E.

```
// File: def/beamformer
//                  Direc                Array_       Elem_
// Key Word Port    tion     Type        Size         Size     Options
// ------------------------------------------------------------------------
   PORT     gain     INPUT   REPLICATED [1][1024]    4
   PORT     ctl      INPUT   CONTROL
   PORT     d_in     INPUT   STRIPED    [100][1024]  8
   PORT     d_out    OUTPUT  STRIPED    [100][512]   8

// File: def/hydrophone
//                  Direc                Array_       Elem_
// Key Word Port    tion     Type        Size         Size     Options
// ------------------------------------------------------------------------
   PORT     elem     OUTPUT  STRIPED    [256][100]   8

// File: def/ship
//                  Direc                Array_       Elem_
// Key Word Port    tion     Type        Size         Size     Options
// ------------------------------------------------------------------------
   PORT     ctl      OUTPUT  CONTROL
   PORT     gain     OUTPUT  REPLICATED [1][1024]    4

// File: def/display
//                  Direc                Array_       Elem_
// Key Word Port    tion     Type        Size         Size     Options
// ------------------------------------------------------------------------
   PORT     elem     INPUT   STRIPED    [4][512]     8

// File: def/matched_filter
//                  Direc                Array_       Elem_
// Key Word Port    tion     Type        Size         Size     Options
// ------------------------------------------------------------------------
   PORT     d_in     INPUT   STRIPED    [100][2048]  8        STRIPED_OVLP = 4
   PORT     gain     INPUT   REPLICATED [1][1024]    4
```

Figure 10. Possible Program Definition files for the receiver system of figure 8.

3.3 DATABASE STARTUP FILE

The SPE provides a global database to store symbolic names with their associated values. Programs are able to use the database to store such things as signal processing parameters, function control and switches, display parameters and control flags, and report and logging flags. Typically these values are found in include files and are shared among programs. If instead they are stored in a global database, then when the values are changed or new ones added, entire sets of programs need not be recompiled.

The SPE allows the user to provide to spe a list of Database Startup files, which contain an initial set of symbolic names and associated values for the system. Symbolic names and their associated values can be assigned to different programs or to specific instances of programs. For example, one may want to set a debugging or logging flag for a specific instance of a particular program, or may

want to set a program variable to different values for each use of the program (i.e., a program that can do multiple functions).

A program can also initialize data in the database from the program interface. However, unlike Database Startup files, a program cannot assign a variable to a specific program or instance of a program.

The database manager is notified that a program will use a variable when the program *registers* for the variable. When a variable in the database is given a value at program startup, the value is propagated automatically to all programs that have registered to use that variable. This means that programs within a system can be developed without the programmer having to know the requirements of other programs. Details about the program interface are discussed in Section 4.3.

The Database Startup files contain a list of variables and associated values used by different programs in the system. Because a system is a set of programs, and each program is a set of instances, a symbolic name can have a different value for each program and instance in a system. Each line in the Database Startup file allows a variable to be assigned to all instances of a specific program, to a specific instance of a program, or to all instances of all programs. The same variable can be specified more than once (on a different line). An example of a Database Startup file is given in figure 11.

```
// File: database/receiver
//
// Key Word Name                Type(Value)        Program(instance)
// ------------------------------------------------------------------------------
   VAR      number_of_widgets   100                display             //integer
   VAR      narrow_band         TRUE               matched_filter      //integer
   VAR      speed_of_sound      1500.0                                 //floating-point
   VAR      input_filename      "sea_test1"        hydrophone          //string
   VAR      debug_stuff         OFF                beamformer          //report
   VAR      interesting_vars    FRAMES,gain,2,5    beamformer(0)       //report
```

Figure 11. Example of a Database Startup file.

The type of value that can be assigned to database variables are *integer*, *real*, *string*, and *report*. Integer database variables may also be specified as `TRUE` or `FALSE`. *Report* variables are a special type of variable that are specified either as `ON` or `OFF`, or as a list of four components: `FRAMES`, *portname*, *startframe*, and *endframe*. The *report* type is explained in more detail in Section 4.4. With the exception of the special *report* type, structures or arrays cannot be assigned through the user interface to variables in the database.

The rules for making a Database Startup file are as follows:

1. Reserved words that can be used in Database Startup files include `VAR`, `TRUE`, `FALSE`, `ON`, `OFF`, and `FRAMES`.
2. Each line starting with `VAR` declares and initializes a variable to be put in the SPE Database. The first and second fields specify the variable name and value. The value specified must be of the type integer, real, string, or report. The last field optionally specifies the program or program and instance the variable is intended for. It may be left empty, indicating that the variable is intended for everyone; it may contain a program name, indicating that the variable is intended for all instances of a program; or it may contain the specific instance of a program for which the variable is intended. If a

program name is given that does not match those used in the System Definition file, the SPE provides a warning message.

3. If the variable is a report variable (see Section 4.4), then the value field is one of `ON`, `OFF`, or `FRAMES`. If the value is `OFF`, then *spe_report()* calls that refer to that variable will not generate output. If the value is `ON`, then *spe_report()* calls that refer to that variable will generate output for the selected program and instance each time it is called. If the value is `FRAMES`, then *spe_report()* calls that refer to that variable will generate output between the times determined by the *portname*, *startframe*, and *endframe* fields, where *portname* is the name of a port for the target program, and *startframe* and *endframe* specify the message counts that delimit the time the report will be generated. Frame numbers start with 1 and increase. Figure 18 contains an example usage of a report variable specified with `FRAMES`. An undefined report variable is considered to be `OFF`.

4. PROGRAMMING INTERFACE

The programmer makes use of SPE features by calling SPE library routines from within the application programs. This section describes the use of those library routines, grouped according to the features provided by the routines: the *message* routines, the *database* routines, the *report* routines, the *memory allocation* routines, the *performance monitoring* routines, and *synchronizing operations*.

4.1 MESSAGE INTERFACE

4.1.1 *spe_init(), spe_send(), spe_recv(), spe_port_id(), spe_port_info()*

The first SPE routine called by the program must be *spe_init()*. This routine blocks until the calling program receives configuration information from the SPE loader. This information includes port interconnections, program parameters, and database values specific to the program instance. The *spe_init()* call determines and allocates the resources needed to perform the message-passing operations used later in the program.

Messages are passed between programs with the *spe_send()* and *spe_recv()* system calls. These calls perform the special scatter and gather operations needed to transfer data between programs with multiple instances. With these calls, each instance of a program will send or receive striped or replicated portions of the data (see Section 2.1). There are also *window* versions of these calls, *spe_send_window()* and *spe_recv_window()*, which provide the user more flexibility in copying out of and into data buffers.

Programs communicate through ports, avoiding the need for the code of a program to contain explicit information on where it is sending or receiving its data. The *spe_send()* and *spe_recv()* system calls require the caller to provide the *port ID* of a port to send or receive data. The port ID of a named port is returned by the *spe_port_id()* system call. Port IDs are assigned by the SPE and must be used when referring to a port.

The portion of the problem that an instance of a program works on can be found from the *spe_port_info()* system call. The *spe_port_info()* routine copies to the supplied address information describing the portion of data that is striped or replicated for the given port and instance. The calling program instance uses this information to determine the portion of data it will work on and to allocate buffers for receiving or sending the data. The calling program must be written so that each instance of it can work on any contiguous-row portion of the data.

These routines and *spe_db_wait()*, which will be described later, represent the minimum set of routines that must be used by a program. An example program using each of these routines is shown in figure 12.

4.1.2 *Message Interface Example*

The program illustrated in figure 12 repeatedly performs FFTs on blocks of input data. The input data blocks can be of any size, but must remain fixed over time. Each block of input data is received on port "in", and each transformed block of output data is sent to port "out". The program repeats itself forever until the SPE system shuts down.

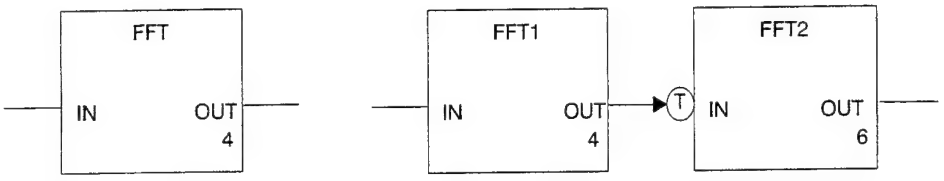
The program is written so that it can be implemented over any number of instances. Resources, such as the buffer space used to receive input messages, are allocated at run time. Careful use of the *spe_port_info()* routine is critical to developing a flexible general-purpose program. The program is

```
/* File: fft.c
 *
 * Description: Performs FFTs on rows of input matrix (rows x columns).
 *              The FFT size is equal to the number of columns in the matrix.
 *              The rows of the input matrix are striped over the program
 *              instances.  For example if the input matrix is 100 x 128 then
 *              a 128-pt FFT will be performed on each row of the matrix.
 */

#include <spe.h>

long          ii, num_rows, fft_size, port_id, in_pid, out_pid;
COMPLEX      *buffer;
size_t        buffer_size;
SPE_PORT_INFO in_port_info;
SPE_STATUS    status;

void main()
{
    /* Initialize the SPE interface */
    spe_init();

    /* Register and assign database variables here. */

    spe_db_wait(); /* Explained in "Database Interface" */

    /* Get the port IDs of ports "in" and "out" */
    in_pid = spe_port_id("in");
    out_pid = spe_port_id("out");

    /* Determine what portion of the problem this instance will work on. */
    spe_port_info(in_pid, &in_port_info);

    num_rows = in_port_info.end_row -
               in_port_info.start_row + 1;
    fft_size = in_port_info.num_columns;

    /* allocate space for the input data. */
    buffer_size = (size_t)(num_rows * fft_size * sizeof(COMPLEX));
    buffer = spe_malloc(buffer_size, "main buffer");

    /* Loop forever until some other program terminates the run. */
    while (1)
    {
        spe_recv(in_pid, buffer, buffer_size, &status);

        for (ii = 0; ii < num_rows; ii++)

            cfft(buffer+ii*fft_size, fft_size, 1); /* buf,size,1=forward */

        spe_send(out_pid, buffer, buffer_size);
    }
}
```

Figure 12. Example FFT program illustrating the use of the basic SPE routines.

written so that it can work on any size data block (rows vs. columns), thus maximizing the reuse of the software.

Figure 13 shows how quickly an application can be built by reusing software. The two-dimensional FFT in figure 13b was constructed by simply connecting two copies of the one-dimensional FFT of figure 13a through a transposed connection. No new software was developed.



(a) One-dimensional FFT. (b) Two-dimensional FFT.

Figure 13. Reuse of an SPE program.

4.1.3 *spe_msg_wait(), spe_msg_wait_list(), spe_probe(), spe_probe_list()*

In the example FFT given in figure 12, the program waits for data on a single port. To wait for data from multiple input ports when the order of the messages is not known ahead of time, the program must use the *spe_msg_wait()* and *spe_probe()* routines. The *spe_msg_wait()* routine blocks until a message is ready to be received on one of the input ports. When a message is available, the *spe_msg_wait()* routine returns with the port ID of the oldest pending message (the message that was available first).

The *spe_probe()* routine determines whether a message on any input port is ready to be received. *spe_probe()* returns immediately with either the port ID of the oldest pending message or the predefined constant SPE_NULL_PORT if there is no message available to be received.

There are extended versions of both *spe_msg_wait()* and *spe_probe()*: *spe_msg_wait_list()* and *spe_probe_list()* allow the user to specify a list of port IDs that should be checked.

The example program illustrated in figure 14 shows how one might use the *spe_msg_wait()* routine. It is not known ahead of time the order in which messages become available over ports "in1" and "in2". However, it is known ahead of time that when a message becomes available on port "in2", another will soon follow on port "in3" (for instance, these messages may be sent from the same program). In figure 14, the *spe_msg_wait()* routine is used to block the program until a message is available to be received, and then *if-else* statements are used to determine from which port

```
/* Get the port IDs of each input port. */
in1_pid = spe_port_id("in1");
in2_pid = spe_port_id("in2");
in3_pid = spe_port_id("in3");
...
while (1)
{
    /* Wait on a message from any input port. */
    pid = spe_msg_wait();

    if (pid == in1_pid)
    {
        spe_recv(in1_pid, buffer1, buffer1_size, &status);
        ...
    }
    else if ( pid == in2_pid)
    {
        spe_recv(in2_pid, buffer2, buffer2_size, &status);
        spe_recv(in3_pid, buffer3, buffer3_size, &status);
        ...
    }
    else ...
}
```

Figure 14. Usage of *spe_msg_wait()*.

to get the message. When a message becomes available on port "in1", it is received and processed. When a message becomes available on port "in2", it is received and processed along with the message from port "in3". The program does not necessarily have to receive and process messages in the order in which they become available. In the period that the messages on ports "in2" and "in3" are received, a message on port "in1" may have become available.

4.1.4 *spe_port_exists(), spe_port_is_connected()*

In a program that has been designed for general use, it may not be known ahead of time whether all ports are actually connected. The routines *spe_port_exists()* and *spe_port_is_connected()* can be used to determine these qualities. The *spe_port_exists()* routine returns a Boolean value indicating whether the named port exists. The *spe_port_is_connected()* routine returns a boolean value indicating whether the named port is connected. Both routines must be supplied with the string name of the port of interest. Data sent to a disconnected port will be silently ignored unless it is being dumped to a file. An attempt to receive data on a port that is not connected will cause the SPE to terminate and produce an error message.

4.1.5 *spe_eos()*

A program can send to an output port an end-of-stream (EOS) mark, indicating that the program will temporarily or permanently stop the flow of data to that port. The EOS mark can be used, for instance, to determine when a system is finished processing or to reroute the flow of data through a system. When sent to a replicated or striped port, the EOS mark also specifies the number of rows and columns of data remaining in the data stream. The EOS mark and the number of rows and columns of valid data can be detected by a receiving program from the *status* argument of the *spe_recv()* routine.

The EOS mark must be used at the end of a stream of data sent to a replicated or striped port to guarantee that the complete data stream can be received. Without the EOS mark, the user cannot receive data from a port's input FIFO if not enough columns have accumulated to make a complete input message. This condition can occur when the number of columns on an input and output connection are different or when block overlap is used at the receiving port.

The entry for *spe_eos()* in Appendix C contains additional details on its use.

Figure 15 shows how a typical data-flow system might be connected. Program A reads data from an input file, program B processes the data, and program C writes the processed data to an output file. Each program executes a loop that receives, processes, and produces frames of data. The system will run until program C writes to the output file the last frame of data that program A produces and program B processes. When program C writes the last frame of data to the output file, it will then initiate system termination, causing all the programs to exit.

For this to happen, program C must be able to determine when it has received the last data that it will write to disk. If this information is not embedded in the data, then it must use some out-of-band technique to determine the end of the data. It is possible to use a separate control port to solve this problem, but this solution often has synchronization problems. For this reason, the EOS mark is provided to indicate that the end of a stream has been reached. It provides a way to tell the user that the last piece of data has been read.

The EOS mark is used as follows: When program A has finished reading the input file and has assembled the last part of the data for program B, it sends an EOS mark to program B by calling

spe_eos(). Program B detects the EOS mark and the number of valid rows and columns from the status information returned by the *spe_recv()* call and, in turn, calls *spe_eos()* to send the EOS mark to program C. Finally, program C detects the EOS mark from the status information returned by the *spe_recv()* call, closes the output file, and initiates system shutdown.

There is an additional example of the use of *spe_eos()* in Appendix C.

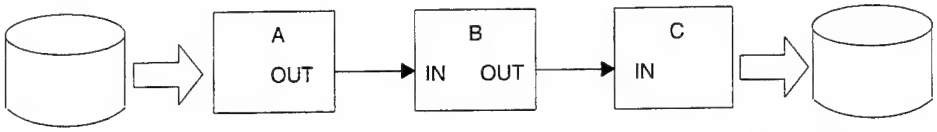


Figure 15. EOS is daisy-chained through programs A, B, and C.

4.1.6 *spe_enter_seq(), spe_leave_seq()*

The *spe_enter_seq()* and *spe_leave_seq()* routines allow the SPE to maintain message order when a program that sends data on one or more control sequence ports also must send data on other kinds of ports. These two classes of ports have a fundamental difference in their method of use. With non-control-sequence ports (replicated, striped, or regular control ports), each instance of a program must access the ports in the same order and frequency. With control sequence ports, each instance of a program is free to use the port(s) as many times as required or not at all, regardless of what occurs in the other instances. To maintain message order without the *spe_enter_seq()* and *spe_leave_seq()* routines, the SPE would have to make a barrier call internally each time a noncontrol-sequence port was called, a possibly expensive operation.

Before an instance of a program can send data to a control sequence output port, each instance of the program must call *spe_enter_seq()*. This routine blocks until all instances of the program have called the routine (i.e., a barrier call) and then causes the SPE to enter a state that allows instances of the program to asynchronously send data to the program's control sequence ports. While in this state, the program cannot send data to striped ports, replicated ports, or regular control ports. Each instance of a program must call *spe_enter_seq()* even if it does not send data to a control sequence port.

When the program is done sending data to its control sequence ports and wishes to send data to its replicated ports, striped ports, or regular control ports, it must call *spe_leave_seq()*. This routine blocks until all instances of the program have called the routine (i.e., a barrier call) and then causes the SPE to enter a state that allows instances of the program to send data to the program's striped, replicated, or regular control ports. While in this state the program cannot send data to its control sequence ports. Each instance of a program must call *spe_leave_seq()* even if it did not send data to a control sequence port.

4.2 TERMINATING SPE PROGRAMS

Because a program is only one part of an SPE application, the correct procedure for terminating or exiting a program used with the SPE is not the same as for a single sequential program. When used with the SPE, *a program should never exit*. There are several reasons for this, primarily because the SPE sends internal messages back and forth between programs. If a program instance exits, then the operating system removes the program's code, and there will be no response to these messages causing other parts of the application to hang. Secondly, even if the user's code has finished its part of the application, the user may wish to interrogate the program through the interactive user interface or to

print performance information collected during the program's execution, and it is not possible to communicate with the program once it has exited.

The SPE provides the routines *spe_idle()* and *spe_terminate()* to correctly handle program and application termination.

4.2.1 *spe_idle()*

If one of the programs in an application has completed its work, but others may still be working, then *spe_idle()* should be called. When a program calls *spe_idle()*, it goes to sleep until the system terminates (e.g., it never returns from the call). This allows a program to stop running without causing other programs to become hung. An idle program will still be able to report results collected from its performance monitors when the program is terminated, as well as to respond to the interactive user interface. An endless program loop that contains no calls to the SPE library will prevent the program from being properly terminated by the SPE.

4.2.2 *spe_terminate(), spe_terminate_define()*

Any program may call *spe_terminate()* if it wishes to terminate the entire application. The SPE will cause each program in the application to terminate when the next SPE routine is called, or if the program is already executing in an SPE routine, to terminate immediately. The SPE does not interrupt what the user's program is currently doing. When notified to terminate, each program will execute an optionally defined user termination routine that is specified using *spe_terminate_define()*. This allows a program to execute critical cleanup code (such as closing files) when the program is terminated by some other program. After executing the user's termination routine, the SPE will generate any *spe_report()* summaries that have been requested and will then exit.

4.3 DATABASE INTERFACE

As described earlier, the SPE provides a global database to store symbolic names with their associated values. Variables can be stored into and read from the database through either a user or program interface. This section describes the program interface.

The program interface, unlike the user interface, does not consider a variable to have a specific destination. That is, a program cannot specify that a variable should contain different values for different programs or instances of programs. This is consistent with the SPE philosophy that a program need not contain knowledge of the existence or requirements of other programs in a system.

A program interfaces to the SPE database by first *registering* each variable that it will access in the database. When a program sets a variable's value in the database, that value propagates to all programs that have registered for it.

4.3.1 *spe_db_register(), spe_db_set(), spe_db_wait()*

Three routines are used by a program to interface to the global database. A program calls the *spe_db_register()* routine to tell the database manager that it is interested in a variable. The program supplies to the routine a string containing the name of the database variable, an address in memory where the local copy of the variable will be maintained, an enumeration indicating the type of variable that it expects, and the size of the variable in bytes.

A program sets the value of a database variable by calling the *spe_db_set()* routine. It supplies to the routine a string containing the name of the database variable, the address in memory where the

value will be copied from, an enumeration indicating the type of variable being stored, and the size of the variable in bytes. Possible values for the types of variables allowed are given in Appendix C where *spe_db_register()* and *spe_db_set()* are described.

After a program has registered or set all database variables, it must call the *spe_db_wait()* routine. This routine is a system-synchronizing routine that waits until all programs in the system have also called *spe_db_wait()*, indicating that they too have registered or set database variables. This routine *must* be called by every program regardless of whether or not the program actually registers or sets database variables. At this time the SPE propagates the values of all variables that have been set, either by the user via configuration files or by the program via *spe_db_set()*, to all programs that have registered for the variable. The *spe_db_wait()* routine also updates local copies of database variables.

The example in figure 16 shows how one might use the global database to make the FFT program more general purpose. The program uses the global database variable "forward_fft" to determine whether it should perform a forward or reverse FFT. The variable would be set from a Database Startup file where it could be set differently for each usage of the program. The FFT program (or any program in the system) must make sure that it does not set a value to this database variable, thus propagating the same value to all usages of the program (different executions of the program may be directed, for instance, to do the FFT in the opposite direction).

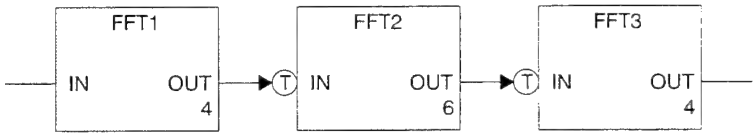
```
void main()
{
    ...
    BOOLEAN       forward_fft = TRUE; /* Default: forward FFT */
    ...
    spe_init();

    /* "forward_fft" will be set in the Database Startup file. */
    spe_db_register("forward_fft", &forward_fft, SPE_DB_INT, sizeof(BOOLEAN));

    /* Wait for other programs to register or set database variables.
     * Update local copies of the database variables. */
    spe_db_wait();
    ...

    while (1)
    {
        spe_recv(in_pid, buffer, buffer_size, &status);

        for (ii = 0; ii < num_rows; ii++)
            if (forward_fft)
               cfft(buffer+ii*fft_size, fft_size, 1); /* Forward FFT */
            else
               cfft(buffer+ii*fft_size, fft_size,-1); /* Reverse FFT */

        spe_send(out_pid, buffer, buffer_size);
    }
}
```

Figure 16. Using a global database variable.

Figure 17 shows how one might use the new FFT program to build a simplified beamformer. Also shown is the Database Startup file that controls whether each program does forward or reverse FFTs. One can see how quickly a system can be built by reusing software.

Extending this concept of reusable software, one might build a general-purpose processing module that could perform any of the functions found in a standard vector-processing library. A global

database variable would determine how each usage of the processing module within a system would function. For instance, the string database variable "libxx_function" could be used to determine if the *libxx.c* program would perform an FFT, correlation, or vector magnitude function. From the Database Startup file, one could specify a different function for each use of the program.



(a) System diagram.

```
// File: database/beamformer
//
// Key Word Name              Type(Value)                    Program(instance)
// ---------------------------------------------------------------------------
   VAR      forward_fft       TRUE                           fft1
   VAR      forward_fft       TRUE                           fft2
   VAR      forward_fft       FALSE                          fft3
```

(b) Database Startup file.

Figure 17. Program reuse controlled by the global database.

4.4 REPORT INTERFACE

Debugging a parallel application requires that the user deal with multiple programs and multiple instances of programs. Using the traditional *printf()* statement to trace the progress of an application is not practical because of its replicated use when called from programs implemented on multiple nodes or from common modules used by multiple programs. When *printf()* is used in a replicated program, the user gets more output than wanted (e.g., 50 repetitions of the same message), and in addition, the user usually does not know from which program or program instance the message has been generated. Also, typically when more than one *printf()* is used simultaneously, their results fragment and mix in the standard output. What is needed instead is a routine that acts like *printf()* but that conditionally executes based on conditions that the user can control.

4.4.1 *spe_report()*

The SPE provides the *spe_report()* call that provides the programmer a way to control and filter debugging and reporting output printing. In addition, all output generated through the *spe_report()* call is recorded into the log file.

The *spe_report()* system call functions the same as the *printf()* system call except that it requires one extra argument. The first argument to *spe_report()* specifies a *report variable* in the global database that *spe_report()* will use at run time to determine if it should actually write the data to standard output. Report variables are created by the user, through Database Startup files, to control which *spe_report()* calls write to standard output. Each use of *spe_report()* can refer to a different report variable, but through careful selection of categories and placement of *spe_report()* calls, one can create an effective debugging environment. The other arguments to *spe_report()* look the same as that used in *printf()*.

The user creates report variables by setting their value in the Database Startup files. Through the use of these variables, the user can specify which categories of reports to display, for which programs

and instances, and for what range of time (time is dictated by range of messages over a specific port). Most of the time, the user will indicate that reports are not desired. *spe_report()* calls that use a report variable that has not been defined will not generate output. Figure 18 is an example of how one might specify a report variable in a Database Startup file.

```
// Key Word Name              Type(Value)          Program(instance)
-------------------------------------------------------------------
     VAR   interesting_vars   FRAMES,gain,2,5      beamformer(0)
```

Figure 18. Specifying *spe_report()* output using `FRAMES` mode.

This line says that we want to see output referring to "interesting_vars" from the *spe_report()* routines called within instance 0 of the beamformer program and called between the times that the gain port receives its second and fifth message. The user can set a different value for "interesting_vars" for each program and instance in the system, or the same value to all instances of a specific program, or the same value to all instances of all programs.

For this report variable to be effective, the beamformer program would contain *spe_report()* calls after places where these interesting variables are computed. For example, a portion of the beamformer program might look like:

```
speed_of_sound = ...
spe_report("interesting_vars","speed_of_sound=%f",speed_of_sound);
...
num_bad_sensors = ...
spe_report("interesting_vars", num_bad_sensors=%d", num_sensors);
```

As a result, when these calls are executed, meeting the conditions specified in the report variable, the *spe_report()* will generate output. For example, the following output might appear:

```
REPORT:beamformer(0):gain:frame=2, interesting_vars, clk=87.887,node=24
-----------------------------------------------------------------------
speed_of_sound=1588.1

REPORT:beamformer(0):gain:frame=2, interesting_vars, clk=87.889,node=24
-----------------------------------------------------------------------
num_bad_sensors=0
```

When the *spe_report()* routine writes data to standard output, it provides a header portion that indicates the name of the report variable, the name of the calling program, the instance of the calling program, the time at which it occurred, and the physical node number. This information is also recorded identically in the SPE log file.

4.4.2 *spe_report_enabled()*

There are times when information to be reported is not readily available, requiring extra manipulation or calculations. In order to avoid this extra work when the report variable is not turned on, the user may call the Boolean function *spe_report_enabled()* to determine if a report call will actually result in any output being produced. As an example:

```
/* avoid extra work if the report variable is not turned on */
if (spe_report_enabled("interesting_vars"))
{
   /* calculations for report output */
   spe_report("interesting_vars", ...);
}
```

4.4.3 *Predefined Report Variables*

The report variables "error", "warning", and "info" are predefined, and they are always set to ON. When a program uses them in a *spe_report()* call, it forces the formatted data to be written to standard output. The "error" and "warning" variables also produce a large noticeable banner on the output. The SPE generates a summary report at system termination that indicates how many times the "error" and "warning" variables were used. Shown below are example uses of these categories:

```
/* Force spe_report() to generate output. */
spe_report("info","Beginning Initialization");
...
if (speed_of_sound > 2000)
    spe_report("warning","speed of sound out of range");
...
```

Many of the SPE system calls have associated report variables that can be set by the user. They can be used to determine when system calls are entered and exited, thus tracing the execution of a program. Other report categories can be used to determine when memory is allocated with *spe_malloc()* or to report performance monitoring statistics. Appendix A contains a list of these variables and shows the information they provide. In Appendix C, the description of each library call summarizes the associated report variables.

4.5 MEMORY ALLOCATION INTERFACE

4.5.1 *spe_malloc(), spe_free()*

Within the SPE, memory allocation is done through special SPE library calls. Users should always use *spe_malloc()* and *spe_free()* rather than calling *malloc()* and *free()* directly. The SPE uses *malloc()* and *free()* internally while protecting these calls from interruption by other SPE interrupt routines in the SPE library. The two routines *spe_malloc()* and *spe_free()* work together to maintain memory usage statistics and the purpose, provided by the caller, for which the various memory allocations were needed. These statistics are used to determine current memory usage that is displayed with *spe_report()* messages. Detailed statistics may be viewed at run time using the interactive user interface (see Section 5.4).

4.6 PERFORMANCE MONITORING INTERFACE

4.6.1 *spe_monitor_on(), spe_monitor_off()*

The SPE provides two simple routines, *spe_monitor_on()* and *spe_monitor_off()*, which the user can use to monitor the performance of sections of code within an application program. The user would use these routines to help find bottlenecks within the application and thus optimize the slow programs of an application or reallocate the hardware resources to the application programs.

The *spe_monitor_on()* and *spe_monitor_off()* routines are placed around sections of code for which the programmer wants performance statistics. The programmer supplies a string argument to the

monitor routines that identifies the section of code to be monitored. The monitor routines keep track of how many times each section of code is entered, how much accumulated time is spent in each section, the minimum and maximum times spent in each section, and the accumulated number of operations performed in each section.

At the end of a run, or by using the interactive user interface during execution, the user can view the information recorded by each of the monitors. For an example of how to use the monitors and view the results, see the entry for *spe_monitor_on()* in Appendix C.

4.7 SYNCHRONIZING OPERATIONS

Parallel programming libraries usually include a set of routines for *global* operations. These routines are *barrier* calls, that is, every program in the application must arrive at the same point in the code and make the same call before they can proceed. In the SPE, where applications are composed of multiple heterogeneous programs, these operations will clearly not work properly because the different programs are doing unrelated things at any given time. However, the SPE provides a similar set of library calls that synchronize only the instances within a given program.

4.7.1 *spe_program_sync()*

Under some circumstances, it is necessary to make certain that all the instances of a program have reached a certain point in their processing before any instance proceeds. The SPE provides a way to synchronize all instances of a program using the *spe_program_sync()* call. *spe_program_sync()* blocks each instance of a program until all instances have arrived at the same point, and then returns.

4.7.2 *spe_global()*

This routine allows the programmer to define a global, synchronizing operation. A typical example is to find the maximum value of a set of values distributed across the instances of a program. An argument to *spe_global()* provides the name of a function that will perform a user-defined operation on data contributed by each instance of the program. The function must be able to reduce (combine) data contributed by each node using an operation that is both associative and commutative (e.g., sum, product, maximum, minimum). Each instance blocks when *spe_global()* is called; when all instances have been provided the output values, the call returns.

4.8 VERIFICATION OF SPE ROUTINE CALLING ORDER

A typical SPE application will have multiple programs, with each program performing a different task. Each program will have multiple instances with each instance performing the same algorithm but on different data. With a few exceptions, this is a basic requirement of the SPE. For the SPE to function properly, most of the calls made to the SPE by each instance of a program must be done in the same order with the same arguments. For example, when a program receives and sends data over replicated, striped, or regular control ports, each instance of the program must make the same port calls in the same order (not true for sequential control ports).

The SPE tries, in the least expensive way, to check that this is being done. As SPE routines are called, each instance of a program maintains a *hash value* of the sequence of SPE calls, along with their arguments. When the program calls an SPE routine that requires that data be passed between instances, the current hash value is also sent (piggybacked with the message) and checked. If the hash values between instances of the program are different, the SPE will terminate the application and print a list of the most recent SPE routines called by the instances with different hash values.

The SPE routines that are hashed are *spe_probe()*, *spe_probe_list()*, *spe_recv()*, *spe_recv_window()*, *spe_discard_data()*, *spe_send()*, *spe_send_window()*, *spe_eos()*, *spe_init()*, *spe_program_sync()*, *spe_eos()*, *spe_enter_seq()*, *spe_leave_seq()*, and *spe_global()*. The port ID arguments of these routines are also hashed. The routines *spe_recv()* and *spe_send()* are not hashed if they refer to data sent via a control sequence port since there is no requirement that all instances of a program execute these calls in the same order.

The routines that cross check (between instances) the hash value are *spe_probe()*, *spe_probe_list()*, *spe_discard_data()*, *programsync()*, *spe_eos()*, *spe_enter_seq()*, and *spe_leave_seq()*. If the database variable "spe_hash_check" is `TRUE`, then the routines *spe_recv()* and *spe_recv_window()* will also cross check the hash value.

5. USING THE SPE

This section describes how to create and run application programs that use the SPE. Both calling line options and interactive commands are explained.

5.1 COMPILING AND LINKING AN SPE PROGRAM

The source code of an SPE program must include the *spe.h* definitions file as indicated in the library descriptions in Appendix C. When compiling and linking an SPE program on the Intel Paragon, you must use the *–nx* switch. To understand the effects of this switch, see the Paragon User's Guide manual. When linking an SPE program, you must link in the library *libspe.a*.

For example, the following command line compiles and links the file *myprogram.c* to create an executable file called *myprogram*:

```
% cc -nx -o myprogram myprogram.c libspe.a -lkmath
```

5.2 RUNNING AN SPE APPLICATION

A user loads and runs an SPE application by executing the `spe` program. This program takes as an argument the name of the System Definition file that `spe` will read to start the application. `spe` then begins to load the programs specified in the System Definition file onto the target nodes of the hardware. From the System Definition file, `spe` reads the Program Definition files and builds and downloads a unique port map to each program instance in the system. `spe` then reads the Database Startup files as specified on the command line and in the Program Definition files and initializes the SPE database. Once this is done, the programs are ready to run.

As an example, to run the system described in figure 8 you would execute:

```
%  mkpart -sz 30 mypart
%  spe -pn mypart -on 0 -s system/receiver -d database/receiver
```

5.3 SPE PROGRAM ARGUMENTS

The calling sequence for `spe` is:

```
%  spe -pn partition -on 0 -s sys_def_filename
          [[-d database_startup_filename]...]
          [-l log_filename]
          [-ident] [-identall] [-nocheck] [-noload] [-nolog] [-portmap]
          [[-Dname]...] [[-Dname=def]...]
```

The arguments *–pn partition –on 0* are arguments to the Paragon *application (1)* command, which say that the `spe` program will run on node 0 of partition *partition.* The remainder of the arguments are passed to the `spe` program after it is loaded on node 0.

Here is a description of the program arguments (listed alphabetically) that may be specified when executing `spe`:

–d database_pathname
: the pathname of the Database Startup file to use for this execution. This option may be repeated allowing the use of multiple Database Startup files that are read and processed in the order given. (optional)

–Dname or *–Dname=value*

the given named variables are set for preprocessing by `cpp` allowing additional control at load time over multiple configuration file parameters. This argument may be repeated to set multiple variables. (optional)

–ident

if specified, `spe` will search all binaries specified in the System Definition file for a string matching the pattern '$Header: .* $' and log the resulting output. These strings may be generated automatically by the RCS revision control system. (optional)

–identall

works identically to the *–ident* option except the RCS header strings for the internal SPE library modules, normally suppressed, are also printed. (optional)

–l logfile_pathname

the pathname in which `spe` will log all output from report calls. By default, all report output is logged to the file `report_log.nn` in the current directory. Interactive user input is also echoed into the log. (optional)

–nocheck

if specified, `spe` will not execute the usual check that the user has compiled and linked with an *spe.h* header file and an *libspe.a* library file that are consistent with the currently executing version of `spe`. Normally, `spe` halts with an error message if the consistency check fails. This option can be used to force `spe` to continue to execute in the presence of mismatching versions, but the user must understand the reason for the inconsistency as well as any possible consequences. (optional)

–noload

if specified, `spe` will check all configuration files but will not actually load any programs. This is a quick way to verify that the System Definition file and all Program Definition files are self-consistent. (optional)

–nolog

if specified, no output will be generated to the log file. (optional)

–on 0

specifies that `spe` itself will execute on only the first node in the partition. This argument indicates to the Paragon runtime system that `spe` is to be loaded and executed on node 0 and only node 0; this is also a requirement of the `spe` program. (required)

–pn partition_name

indicates to the Paragon runtime system the partition in which `spe` will execute. (required)

–portmap

if specified, `spe` will log additional detailed information on the port connections between all instances of all programs. (optional)

–s system_definition_pathname

the pathname of the System Definition file to use for this execution. (required)

5.4 INTERACTIVE USER INTERFACE

While an SPE application is running, the user can interact with the SPE to determine a number of things about the status of the application. Here is a description of the available interactive commands (listed alphabetically):

b <node#>
Show why *node#* is blocked.

database or *d*
Print the contents of the database.

help or *h* or *?*
Print this list of interactive commands.

hi <node#>
Print a history of the most recent SPE library calls made on *node#*.

list or *l*
List the nodes that have not terminated or cannot respond to the SPE *stop* command.

ma <node#>
List all monitor values (including detailed SPE monitors) for *node#*. If *node#* is omitted, list them for instance 0 of all programs.

me <node#>
Show how much memory has been allocated on *node#* using the *spe_malloc()* call. This includes memory allocated internally by the SPE.

mesh or *m*
Show a picture of the mesh and the nodes on which programs are running.

mo <node#>
List the monitor values for *node#*. If *node#* is omitted, list the monitor values for instance 0 of all programs.

node_state or *n*
Print a summary of the states of all instances of all programs including in which SPE library routine the program is executing, and how much memory has been allocated.

rm
Reset the monitors on all nodes to zero.

stop or *s*
Stop `spe` and all its programs. Each node initiates termination at the next entry to the SPE library: any user-specified termination function is invoked and the program then exits.

w <node#>
Cause the program on *node#* to print a function stack trace and the contents of the hardware registers.

Appendix A: PREDEFINED REPORTS

The following report variables are predefined by the SPE and produce output messages as described below.

spe_db_wait

```
spe_db_wait()(entering):
spe_db_wait()(exiting):
```

spe_discard_data

```
spe_discard_data("portname")(entering): Waiting to discard data.
spe_discard_data("portname")(exiting): Done discarding data.
```

spe_eos

```
spe_recv("portname")(exiting): EOS detected
      (rows=%d, cols=%d, frame = %d).
spe_recv_window("portname")(exiting): EOS detected
      (rows=%d, cols=%d, frame = %d).
```

spe_idle

```
spe_idle()(entering): Program has gone into idle state.
```

spe_init

```
spe_init()(entering):
spe_init()(exiting): freemem = %d
```

spe_malloc

```
spe_malloc(): Malloced %d bytes of memory, at address = %d for
      "purpose_str".  %d bytes of memory left.
```

spe_msg_wait

```
spe_msg_wait()(entering): Waiting for an available message on any port.
spe_msg_wait()(exiting): Message available on port "portname"
```

spe_msg_wait_list

```
spe_msg_wait_list()(entering): Waiting for an available message on
      selected ports.
spe_msg_wait_list()(exiting): Message available on port "portname"
```

spe_probe

```
spe_probe()(entering): Checking for message on any port.
spe_probe()(exiting): Message available on port "portname".
spe_probe()(exiting): Message not available.
```

spe_probe_list

```
spe_probe_list()(entering): Checking for message on selected ports.
spe_probe_list()(exiting): Message available on port "portname".
spe_probe_list()(exiting): Message not available.
```

spe_program_sync

```
spe_program_sync()(entering): Waiting for instances of program
      to sychronize.
spe_program_sync()(exiting): Instances of program have sychronized.
```

spe_recv

```
spe_recv("portname")(entering): Waiting to receive message (frame %d).
spe_recv("portname")(exiting): Received message (frame %d).
```

spe_recv_window

```
spe_recv_window("portname")(entering): Waiting to receive message
      (frame %d).
spe_recv_window("portname")(exiting): Received message (frame %d).
```

spe_send

```
spe_send("portname")(entering): Waiting to send message (frame %d).
spe_send("portname")(exiting): Sent message (frame %d).
```

spe_send_window

```
spe_send_window("portname")(entering): Waiting to send message
      (frame %d).
spe_send_window("portname")(exiting): Sent message (frame %d).
```

spe_show_monitors

[standard monitor information display is printed]

spe_terminate

```
Starting termination.
Finishing termination.
```

Appendix B: RESERVED WORDS

These words are reserved for SPE use in all configuration files.

```
ALL
ANY
APPEND
ASCII
BLOCK_OVLP
BUFFER
CEIL
CONTROL
CONVERT
DUMP
EXCLUDE
FALSE
FILENAME
FLOOR
FRAMES
INPUT
INT
MATLAB
MAX
MIN
NET
NO_HEADER
OFF
ON
OUTPUT
PORT
PROGRAM
REAL
RENAME
REPLICATED
ROUND_ROBIN
SEQUENCE
SPE
STRIPED
STRIPED_OVLP
STRUCTURE
TRANSPOSE
TRUE
VAR
```

Appendix C: PROGRAMMING CALLS

SPE_CLOCK()

spe_clock(): Returns the elapsed time in seconds since the application started running.

Synopsis

```
#include <spe.h>

double spe_clock(void);
```

Return Value

Returns a double precision value for the elapsed time in seconds since the application started running.

Description

The **spe_clock**() routine measures the time interval in seconds since the application started running. When the SPE starts an application, it sends to each program instance an offset time that the SPE uses (adds the value to **dclock**()) to get the elapsed time since the application started running.

On the Intel Paragon, the time interval has a resolution of 100 nanoseconds.

Errors

None.

Report Variables

None.

SPE_DB_REGISTER()

spe_db_register(): Tell the database manager that we are using a variable of a given name and size.

Synopsis

```
#include <spe.h>

void spe_db_register(
        const char          *name,
        void                *address,
        SPE_DB_TYPE         type,
        size_t              size);

typedef enum SPE_DB_TYPE {
        SPE_DB_INT,
        SPE_DB_FLOAT,
        SPE_DB_DOUBLE,
        SPE_DB_STRING,
        SPE_DB_REPORT,
        SPE_DB_USER_DEFINED
} SPE_DB_TYPE;
```

Parameters

name	is the symbolic name of the variable to be registered. *name* must be 31 characters or less.
address	is the address in memory where the variable will be maintained.
type	is an enumeration indicating the type of variable expected.
size	is the size in bytes of the variable to be maintained.

Description

Tell the database manager that a variable of the given *name* and *size* will be used. Each program that uses a database variable must register for it. All programs registering for the same variable must give the same value for the *type* and *size* parameters. After registering for all database variables used by the program, the program must call **spe_db_wait**(), after which the values of the variables will have been updated from the database. If a variable is not given a value in the database, either by being initialized from the Database Startup file or by being set in a program with **spe_db_set**(), then the value of the variable will not be changed.

Each instance of a program must call **spe_db_register**() with the same arguments.

Normally, the SPE checks the *size* of the variable to verify that it is consistent with the *type* specified. However, if the user gives the type as **SPE_DB_USER_DEFINED**, this check is skipped and the user can specify a variable of any size.

Errors

The SPE will terminate and produce an error message if the *type* or *size* arguments disagree with what is stored in the database.

Report Variables

none

SPE_DB_SET()

spe_db_set(): Copy the value at the specified address to the named variable in the database.

Synopsis

```
#include <spe.h>

void spe_db_set(
        const char          *name,
        void                *address,
        SPE_DB_TYPE         type,
        size_t              size);

typedef enum SPE_DB_TYPE {
        SPE_DB_INT,
        SPE_DB_FLOAT,
        SPE_DB_DOUBLE,
        SPE_DB_STRING,
        SPE_DB_REPORT,
        SPE_DB_USER_DEFINED
} SPE_DB_TYPE;
```

Parameters

name	is the symbolic name of the database variable to which a new value will be copied. *name* must be 31 characters or less. The variable must have already been registered with **spe_db_register()**.
address	is the address in memory where the value is copied from.
type	is an enumeration indicating the type of variable being stored to the database.
size	is the size in bytes of the variable to be copied. If the size does not agree with the registered variable, then the SPE system will terminate and produce an error message.

Description

Copy the value at the specified address to the named variable in the database. See **spe_db_register()** for a description of database operation.

Each instance of a program must call **spe_db_set()** with the same arguments.

Errors

The SPE will terminate and produce an error message if the *type* or *size* arguments disagree with what is stored in the database.

Report Variables

none

SPE_DB_WAIT()

spe_db_wait(): Wait until all programs in the system have registered and set database variables.

Synopsis

```
#include <spe.h>

void spe_db_wait(void);
```

Description

Wait until all programs in the system have registered and set database variables. This routine is used as a form of synchronization to the database to make sure that all programs have registered variables and have the correct values before proceeding. Every program must call **spe_db_wait**() whether or not any database variables are used.

Errors

Must be called only once and after **spe_init**() or else the SPE will terminate and produce an error message.

Report Variables

spe_db_wait

SPE_DISCARD_DATA()

spe_discard_data(): Discard columns of input data on a port.

Synopsis

```
#include <spe.h>

void spe_discard_data(
        long        port_id,
        size_t      num_columns);
```

Parameters

port_id is the port ID of the input port on which data is to be discarded.

num_columns is the number of input columns of data to discard.

Description

Discard the given number of columns of input data on the specified port. The call does not block. If the requested number of columns is not currently available, they will be discarded when they eventually arrive.

Each instance of a program must call **spe_discard_data**() with the same arguments.

Restrictions

This call may only be used in very limited circumstances. Discarding data is only done at the very start of a data stream. It must be called just once before any call to receive data and will fail to work properly if an EOS mark is encountered on the port before the designated number of columns has arrived.

Errors

The *port_id* argument must refer to a valid, noncontrol input port or else the SPE will terminate and produce an error message. The port must also be connected. The *num_columns* argument must be greater than zero and less than or equal to the number of columns specified for the port.

Report Variables

spe_discard_data

SPE_DUMP_DEFINE()

spe_dump_define(): Specifies a data format conversion function to be executed when port data are being dumped.

Synopsis

```
#include <spe.h>

void spe_dump_define(
        long    port_id,
        void    (*dump_function) (void *src, void *dst, long choice));
```

Parameters

port_id is the port ID of the port on which data are to be dumped.

dump_function is the name of the function to execute when the data for *port_id* are being dumped and a format conversion is required. The function must have three arguments as described below and return no value.

src is the address of the source data element to be converted.

dst is the address of the output data element after conversion.

choice is an integer that is passed to the dump conversion routine from the System Definition file where the dump is specified.

Description

Specifies a data conversion function to be executed when port data are being dumped and the element being dumped must be converted from one format to another. The *choice* argument to the conversion function passes on to the conversion routine an integer from the DUMP specification in the System Definition file, thereby allowing the user to select from more than one format conversion possibilities on a given port. If the *choice* field was omitted in the dump specification, the value passed is 0.

Errors

None.

Report Variables

None.

SPE_ENTER_SEQ()

spe_enter_seq(): Allow access to control sequence ports.

Synopsis

```
#include <spe.h>

void spe_enter_seq(void);
```

Description

Before an instance of a program can send data to a control sequence output port, each instance of the program must call **spe_enter_seq()**. This routine blocks until all instances of the program have called the routine (i.e., a barrier call) and then causes the SPE to enter a state that allows instances of the program to asynchronously send data to the program's control sequence ports. While in this state, the program cannot send data to striped ports, replicated ports, or regular control ports. Each instance of a program must call **spe_enter_seq()** even if it does not send data to a control sequence port.

The **spe_enter_seq()** and **spe_leave_seq()** routines allow the SPE to maintain message order when a program must send data to both control sequence and noncontrol sequence ports. These two classes of ports have a fundamental difference in that with noncontrol sequence ports (replicated, striped, or regular control ports), each instance of a program must access them in the same order and frequency whereas with control sequence ports, each instance of a program is free to use the port(s) as many times as it wants or not at all, regardless of what the other instances are doing. To maintain message order without the **spe_enter_seq()** and **spe_leave_seq()** routines, the SPE would have to make a barrier call internally each time a noncontrol sequence port was called, a possibly expensive operation.

Errors

The SPE will terminate and produce an error message if the code is already running within a sequential area.

Report Variables

None.

SPE_EOS()

spe_eos(): Sends an end-of-stream (EOS) mark to an output port.

Synopsis

```
#include <spe.h>

long spe_eos(
        long        port_id,
        long        rows,
        long        columns);
```

Parameters

port_id is the port ID of the output port to which the EOS mark will be sent.

rows is the number of rows of data remaining before the EOS mark (may be zero). **spe_eos**() ignores the *rows* value if *port_id* specifies a control port.

columns is the number of columns of data remaining before the EOS mark (may be zero). **spe_eos**() ignores the *columns* value if *port_id* specifies a control port.

Description

spe_eos() sends a transparent EOS mark to a specified output port. The EOS mark indicates the program will temporarily or permanently stop the flow of data to the port. The EOS mark and the number of valid rows and columns of data at the end of the stream can be detected by a receiving program from the status argument of the **spe_recv**() routine.

The EOS mark must be used at the end of a stream of data sent to a replicated or striped port to guarantee that the complete data stream can be received. Without the EOS mark the user cannot receive data from a port's input FIFO if not enough columns have accumulated to make a complete input message. This condition can occur when the number of columns on an input and output connection are different, or when block overlap is used at the receiving port.

When **spe_eos**() is used on a replicated or striped port and both the *rows* and *columns* arguments are non-zero, then the EOS mark occurs within the data sent in the next **spe_send**() call. The *rows* and *columns* values indicate the amount of data remaining in the stream. One of the two values must be set to its corresponding value found in the Program Definition file, while the other value may be truncated in the range from 1 to its corresponding value. Valid data are contained within the rows 0 to *rows–1* and columns 0 to *columns–1*. An entire buffer may be valid. The second **spe_send**() call following the **spe_eos**() call will contain the beginning of the next stream of data.

When **spe_eos**() is used on a control port, or on a replicated or striped port when either the *rows* or *columns* argument is set to zero, then the EOS mark is inserted between the previous and next frame of data sent to the specified output port. The next **spe_send**() call after the **spe_eos**() call will contain data for the beginning of the next stream of data.

Each instance of a program must call **spe_eos**() with the same argument values.

Restrictions

When EOS is used with nontransposed replicated or striped connections, the *rows* value can be truncated only if, in the Port Definition files, the number of columns on the output and input port are equal, and block overlap is not specified.

When EOS is used with transposed replicated or striped connections, the *columns* value can be truncated only if, in the Port Definition files, the number of rows on the output port and the number of columns on the input port are equal and block overlap is not specified.

spe_eos() cannot be used on a *control-sequence* output port.

spe_discard_data() cannot be used on an input port that will get an EOS mark on the first **spe_recv**().

Errors

The SPE will terminate and produce an error message if the *port_id* argument is not a valid output port ID or the *rows* or *columns* arguments have values out of range.

Report Variables

spe_eos

Example

```
-----------------------------------------------------------------
/* Program which generates data. */
...
spe_port_info(out, &info);
rows_in_pkt = info.no_rows;
columns_in_pkt = info.no_columns;

for (stream = 0; stream < num_streams; stream++)
{
    columns_left_in_stream = num_columns_in_stream;
    while (columns_left_in_stream)
    {
        /* make packet of data */
        ...
        /* If last packet then send EOS mark to port */
        if (columns_left_in_stream <= columns_in_pkt)
        {
            spe_eos(out, rows_in_pkt, columns_left);
            spe_send(out, buf, size);
            columns_left_in_stream = 0;
        }
        else
        {
            spe_send(out, buf, size);
            columns_left_in_stream -= columns_in_pkt;
        }
    }
}
```

```
spe_idle();
```

```
/* Program which filters data. */

...
while (1)
{
    spe_recv(in, in_buf, in_size, &status);
    /* status is structure containing EOS flag, valid rows and
     * columns.
     */

    if (status.eos)
        spe_eos(out, status.rows, status.columns)

    if (!status.eos || status.columns != 0)
    {
       /* process data */
       ...
       spe_send(out, out_buf, out_size);
    }
}
```

SPE_FREE()

spe_free(): Releases memory just like **free**() but also maintains SPE statistics for memory usage.

Synopsis

```
#include <spe.h>

void *spe_free(
        void            *buf);
```

Parameters

buf is the address of a buffer that was previously allocated with **spe_malloc**().

Description

Frees memory allocated with **spe_malloc**() using **free**() while protecting the **free**() call from interruption by other SPE interrupt routines in the SPE library. Users should always use **spe_free**() rather than calling **free**() directly. The **spe_free**() library call maintains the internal SPE statistics on memory usage. See **spe_malloc**() for details.

Errors

None.

Report Variables

None.

SPE_GLOBAL()

spe_global(): Performs a user-defined global operation across all instances of the program.

Synopsis

```
#include <spe.h>

void spe_global(
        void        (*user_function) (void *src1, void *src2, void *dst1),
        void        *src,
        void        *dst,
        long         size);
```

Parameters

user_function is the name of the user-defined function that will be called to perform work on data collected by **spe_global()**. *user_function()* has two input (source) pointers and one output (destination) pointer.

src is a pointer to the source of data provided by each instance of the program.

dst is a pointer to the buffer where the results of the global operation are stored.

size is the number of bytes of data in the source and destination buffers.

Description

The **spe_global()** function performs the user-defined global operation on data contributed by each instance of the program. The user-defined function must be able to reduce (combine) data contributed by each node using an operation that is both associative and commutative (e.g., sum, product, maximum). Each instance of the program is returned the same results from the global operation.

The **spe_global()** function arranges that the user-defined function is called repeatedly so that the global data are reduced in a pair-wise fashion across the instances of the program. **spe_global()** communicates source data and results from previous reductions between instances so that the final results can be reduced into one instance and then passed to the other instances of the program. The order in which the pair-wise reductions of data is performed is implementation dependent.

The user-defined function must be able to reduce data pointed to by *src1* and *src2* and store the results to *dst1*. The function must have no other side effects since on any given program instance, the SPE may call the user-defined function once, several times, or not at all.

If a single-instanced program calls **spe_global()**, then the user-defined function will be called once with *src1* set to *src*, *src2* set to NULL, and *dst1* set to *dst*. It will be left up to the user-defined function to copy the contents of *src1* to *dst1* if that is the desired effect (e.g., maximum, minimum, sum).

When **spe_global()** is called, it allocates an internal working space of three times *size* bytes. This space is freed before returning to the user.

Each instance of a program must call **spe_global()**.

Errors

The SPE will terminate and produce an error message if all instances do not provide the same user-defined function and the same *size* argument.

Report Variables

None.

Examples

This example illustrates an implementation of the common global operation of finding the sum of the values in a distributed vector. This example assumes that each program instance contains only a single value to be summed. The user-defined function `sum()` simply adds the two input values.

```
-----------------------------------------------------------------
long myvalue
long total;

void sum(long *src1, long *src2, long *dst)
{
    if (src2 == NULL)
        /* this is the only instance so copy input to output */
        *dst = *src1;
    else
        *dst = *src1 + *src2;
} /* end sum() */

...
spe_global(sum, &myvalue, &total, sizeof(long));
/* total now contains sum of all values across all instances */
-----------------------------------------------------------------
```

In the following example, each program instance contains an array of values that is to be summed columnwise across all instances. The user-defined function `sumv()` adds a series of input values, the number of which is given by the global variable `num_elts`. The result of the call to **spe_global()** is an array in which each element contains the sum of the corresponding elements from all instances.

```
-----------------------------------------------------------------
long num_elts = 3; /* global to communicate with sum routine */
long input[3], output[3]; /* arrays of length num_elts */

void sumv(long *src1, long *src2, long *dst)
{
    long ii;
    if (src2 == NULL)
        /* this is the only instance so copy input to output */
        for (ii=0; ii < num_elts; ii++)
            dst[ii] = src1[ii];
    else
        for (ii=0; ii < num_elts; ii++)
```

```
            dst[ii] = src1[ii] + src2[ii];
} /* end sumv() */

...
if (mynode = 0)
    {input[0] =  1; input[1] =  2; input[3] =  3;}
else
    {input[0] = 10; input[1] = 20; input[3] = 30;}

spe_global(sumv, input, output, num_elts * sizeof(long));
printf("node %d: %d %d %d\n",
       mynode, output[0], output[1], output[2]);
```

When run with two instances, the output of this example program would be:

```
node 0: 11 22 33
node 1: 11 22 33
```

SPE_IDLE()

spe_idle(): Goes to sleep until the system terminates (never returns).

Synopsis

```
#include <spe.h>

void spe_idle(void);
```

Description

Goes to sleep until the system terminates (never returns). Allows a program to stop running without causing other programs to hang. An idle program will still be able to report results collected from its performance monitors when terminated or requested by the user from the interactive user interface.

Errors

None.

Report Variables

spe_idle

SPE_INIT()

spe_init(): Initializes the SPE interface. Blocks until all programs in an SPE application have called this routine.

Synopsis

```
#include <spe.h>

void spe_init(void);
```

Description

Initializes the SPE interface and must be the first SPE routine called. This routine blocks until all programs have called **spe_init**().

Errors

The SPE will terminate if a program cannot initialize properly.

Report Variables

spe_init

SPE_LEAVE_SEQ()

spe_leave_seq(): Leave control sequence port access code.

Synopsis

```
#include <spe.h>

void spe_leave_seq(void);
```

Description

When the program is done sending data to its control sequence ports and wishes to send data to its replicated ports, striped ports, or regular control ports, it must call **spe_leave_seq()**. This routine blocks until all instances of the program have called the routine (i.e., a barrier call) and then causes the SPE to enter a state that allows instances of the program to send data to the program's striped ports, replicated ports, or regular control ports. While in this state, the program cannot send data to its control sequence ports. Each instance of a program must call **spe_leave_seq()** even if it did not send data to a control sequence port.

See the entry for **spe_enter_seq()** for more information.

Errors

The SPE will terminate and produce an error message if the code is not currently running within a sequential area.

Report Variables

None.

SPE_MALLOC()

spe_malloc(): Gets memory just like **malloc()** but also generates **report()** messages indicating usage.

Synopsis

```
#include <spe.h>

void *spe_malloc(
        size_t      size,
        char        *purpose_str);
```

Parameters

size	is the amount of memory in bytes to allocate.
purpose_str	is the string describing the purpose of the allocation.

Return Value

Returns pointer to space allocated.

Description

Gets memory using **malloc()** while protecting the **malloc()** call from interruption by other SPE interrupt routines in the SPE library. Users should always use **spe_malloc()** rather than call **malloc()** directly. The argument *purpose_str* is a string supplied by the caller indicating the purpose of the allocation. This argument is used in report and statistics messages.

The two routines **spe_malloc()** and **spe_free()** work together to maintain memory usage statistics and the purpose for which the various memory allocations were needed. These statistics are used to determine the current memory usage that is displayed with **spe_report()** messages. Detailed statistics may be viewed through the interactive user interface.

Errors

The SPE will terminate and produce a report message if there is not enough memory to allocate the amount requested.

Report Variables

spe_malloc

SPE_MONITOR_OFF()

spe_monitor_off(): Keep performance statistics on a section of code.

Synopsis

```
#include <spe.h>

void spe_monitor_off(
        const char      *section_name,
        long            num_ops);
```

Parameters

section_name is the name of the section being monitored. Must be the same as used in the corresponding **spe_monitor_on()** call for the section being monitored. *section_name* must be 31 characters or less.

num_ops is the number of operations executed in the section of code being monitored. The user provides this value based on the user's knowledge of the algorithms performed in the monitored section of code. The *num_ops* argument can be set to zero if the user does not care about the operations per second statistic for this section of code.

Description

See the entry for **spe_monitor_on()** for a description of how this routine is used.

Errors

The SPE will terminate and produce an error message if **spe_monitor_on()** and **spe_monitor_off()** are not called in order for a given section of code.

Report Variables

spe_show_monitors

SPE_MONITOR_ON()

spe_monitor_on(): Keep performance statistics on a section of code.

Synopsis

```
#include <spe.h>

void spe_monitor_on(
        const char      *section_name);
```

Parameters

section_name is the name of the section being monitored. Must be the same as used in the corresponding **spe_monitor_off()** call for the section being monitored. *section_name* must be 31 characters or less.

Description

The performance monitoring routines are placed around sections of code for which the programmer wants performance statistics. **spe_monitor_on()** and **spe_monitor_off()** are placed, respectively, at the beginning and end of a section of code. The same *section_name* string must be supplied to both. When the **spe_monitor_on()** routine is called, the time on the hardware clock is recorded for the section of code that will be monitored. When the corresponding **spe_monitor_off()** routine is called (with the same *section_name*), the hardware clock is read, and the elapsed time since the **spe_monitor_on()** routine was called is computed. The elapsed time is added to a variable keeping track of accumulated time, and compared to other variables keeping track of minimum and maximum values. Also recorded are the number of times each section of code is entered and the number of accumulated operations performed by each.

The user can view the data recorded by the monitor routines either interactively during the run, or at the end of the run by turning on the "spe_show_monitors" report variable for the programs and instances of interest.

Sections of code surrounded by the **spe_monitor_on()** and **spe_monitor_off()** routines can be embedded within other sections of code being monitored. Also, different sections of code can use the same *section_name,* thus grouping the statistics for those sections.

Errors

The SPE will terminate and produce an error message if **spe_monitor_on()** and **spe_monitor_off()** are not called in order for a given section of code.

Report Variables

spe_show_monitors

Example

```
...
spe_monitor_on("both");

/* 128-pt Forward FFT */
spe_monitor_on("fft");
```

```
cfft(buf, 128, 1);
spe_monitor_off("fft",4480); /* 5n*logn = 4480 */

/* 128-pt Inverse FFT */
spe_monitor_on("ifft");
cfft(buf, 128, -1);
spe_monitor_off("ifft",4480);

spe_monitor_off("both",0);
...
```

SPE_MSG_COUNT()

spe_msg_count(): Returns the number of times data have been sent or received on a port.

Synopsis

```
#include <spe.h>

long spe_msg_count(
        long            port_id);
```

Parameters

port_id is the port ID of the input or output port for which the user is requesting information.

Description

This routine provides the user a way to find out how many send or receive calls have been made on a given port.

Errors

The SPE will terminate and produce an error message if the port does not exist.

Report Variables

None.

SPE_MSG_LEN()

spe_msg_len(): Returns the length in bytes of the most recently encountered control message.

Synopsis

```
#include <spe.h>

long spe_msg_len(void);
```

Description

This routine provides the user a way to find out the number of bytes in the last message encountered by a call to **spe_probe**(), **spe_probe_list**(), **spe_msg_wait**(), **spe_msg_wait_list**(), or **spe_recv**().

Errors

None.

Report Variables

None.

SPE_MSG_WAIT()

spe_msg_wait(): Waits until a message is ready to be received and returns the port ID for the message.

Synopsis

```
#include <spe.h>

long spe_msg_wait(void);
```

Return Value

Returns the port ID of a message ready to be received.

Description

The **spe_msg_wait**() routine blocks until a message is ready to be received on one of the input ports. Then when a message is available, the **spe_msg_wait**() routine returns with the port ID of the pending port. The **spe_msg_wait**() routine always returns the port IDs of the input messages in the order they were received. All instances of a program are guaranteed to receive the input messages in the same order. See also **spe_msg_wait_list**() and **spe_probe**().

Errors

The SPE will terminate and produce an error message if there are no input ports specified in the System Definition file for the calling program.

Report Variables

spe_msg_wait

SPE_MSG_WAIT_LIST()

spe_msg_wait_list(): Waits until a message on one of the specified input ports is ready to be received and returns the port ID for the message.

Synopsis

```
#include <spe.h>

long spe_msg_wait_list(
        long        port_id_list[],
        long        num_port_ids);
```

Parameters

port_id_list is a list of port IDs that **spe_msg_wait_list**() will wait on for a message ready to be received. **spe_msg_wait_list**() will ignore entries in the port ID list that are set to the predefined constant **SPE_NULL_PORT**. The port ID list must contain at least one non-null entry.

num_port_ids is the number of port IDs (including any null entries) in *port_id_list*. *num_port_ids* must be greater than zero.

Return Value

Returns the port ID of the next message ready to be received, which is in the list of port IDs passed to **spe_msg_wait_list**().

Description

The **spe_msg_wait_list**() routine blocks until a message on one of ports specified by *port_id_list* is ready to be received. When a message becomes available, its port ID is returned. **spe_msg_wait_list**() will ignore entries in the port ID list that are set to the predefined constant **SPE_NULL_PORT**. All instances of a program are guaranteed to receive input messages in the same order. See also **spe_msg_wait**() and **spe_probe**().

Errors

The SPE will terminate and produce an error message if the input port ID list contains a nonvalid input port ID or it does not specify at least one non-null, connected input port.

Report Variables

spe_msg_wait_list

Examples

```
------------------------------------------------------------------
/* Example of how to use spe_msg_wait_list() */
...
data = spe_port_id("data");
cmd1 = spe_port_id("cmd1");
cmd2 = spe_port_id("cmd2");

list[0] = data;
list[1] = cmd2;
```

```
...

/* Wait for message on any port */
pid = spe_msg_wait()
...

/* Wait for message on the "data" or "cmd2" port */
pid = spe_msg_wait_list(list, 2);
```

```
/* Example of how to use SPE_NULL_PORT */
...
data = spe_port_id("data");
cmd1 = spe_port_id("cmd1");
cmd2 = spe_port_id("cmd2");

list[0] = data;
list[1] = SPE_NULL_PORT;
list[2] = cmd2;

/* Wait for message on the "data" or "cmd2" port */
pid = spe_msg_wait_list(list, 3);

/* Now wait for message on only the "cmd2" port */
list[0] = SPE_NULL_PORT;
pid = spe_msg_wait_list(list, 3);
```

SPE_PORT_EXISTS()

spe_port_exists(): Returns a Boolean value indicating whether a port exists.

Synopsis

```
#include <spe.h>

BOOLEAN spe_port_exists(
        char            *port_name);
```

Parameters

port_name is the name of the port to check for existence or connectivity. *port_name* must be 31 characters or less.

Description

spe_port_exists() returns a Boolean value indicating whether the named port exists. This routine can be used by a program designed to work with any number of input or output ports. For example, in a multiplexing program it may not be known ahead of time how many input ports will be used to multiplex the data.

Errors

None.

Report Variables

None.

SPE_PORT_ID()

spe_port_id(): Returns the port ID for the named port.

Synopsis

```
#include <spe.h>

long spe_port_id(
        char            *port_name);
```

Parameters

port_name must be the name of one of the ports specified in the Program Definition file of the calling program. If the named port does not exist, then the SPE will terminate the run and produce an error message. *port_name* must be 31 characters or less.

Return Value

Returns the port ID for the named port.

Description

Returns the port ID for the named port. The SPE library calls use port IDs to receive or send data over the specified ports.

Errors

The SPE will terminate if the named port does not exist in the Program Definition file of the calling program.

Report Variables

None.

SPE_PORT_INFO()

spe_port_info(): Copies the configuration information of a port to an address supplied by the caller.

Synopsis

```
#include <spe.h>

void spe_port_info(
        long                port_id,
        SPE_PORT_INFO       *port_info);

typedef struct SPE_PORT_INFO {
        /* Contains values that are common to each instance. */
        char            name[32];
        SPE_PORT_TYPE   type;
        BOOLEAN         is_input;
        BOOLEAN         is_transposed;
        long            num_buffers;    /* additional buffers requested */
        long            num_rows;       /* across all instances */
        long            num_columns;    /* across all instances */
        long            elem_size;      /* in bytes */
        long            block_ovlp;     /* in columns */

        /* Contains values that are unique to each instance. */
        long            start_row;
        long            end_row;
        long            start_ovlp_row;
        long            end_ovlp_row;
} SPE_PORT_INFO;

typedef enum SPE_PORT_TYPE {
        SPE_REPLICATED, SPE_STRIPED, SPE_CONTROL,
        SPE_CONTROL_SEQUENCE, SPE_CONTROL_ROUND_ROBIN
} SPE_PORT_TYPE;
```

Parameters

port_id	is the port ID of the port for which information is sought.
port_info	is the address to which the port's configuration information will be copied.

Description

Copies the configuration information of a port to the address supplied by the caller. Portions of the configuration information will be unique to the instance of the calling program. The calling program uses this information to determine which portion of the problem it works on. If *type* is **SPE_STRIPED,** then *start_row, end_row, start_ovlp_row,* and *end_ovlp_row* contain valid data that are unique to each instance. They are not used for any other port type. For output **SPE_STRIPED** ports, there is no overlap, so *start_ovlp_row* and *end_ovlp_row* are set equal to *start_row* and *end_row*, respectively. The other fields of the structure always contain valid data and are the same for each instance of a given port.

Errors

The SPE will terminate and produce an error message if the *port_id* argument is not a valid port ID.

Report Variables

None.

SPE_PORT_IS_CONNECTED()

spe_port_is_connected(): Returns a Boolean value indicating whether a port is connected.

Synopsis

```
#include <spe.h>

BOOLEAN spe_port_is_connected(
        char            *port_name);
```

Parameters

port_name is the name of the port to check for connectivity. *port_name* must be 31 characters or less.

Description

spe_port_is_connected() returns a Boolean value indicating whether the named port is connected to a net. This routine can be used by a program designed to allow partial connectivity to its ports. It will allow the program to avoid reading or writing to ports not connected to a net.

Errors

The SPE will terminate and produce an error message if *port_name* is not a valid port.

Report Variables

None.

SPE_PORT_NAME()

spe_port_name(): Returns a pointer to the string name of a port.

Synopsis

```
#include <spe.h>
char *spe_port_name(
        long            port_id);
```

Parameters

port_id must be a valid port ID.

Description

Returns a pointer to the string name of the port for the given port ID.

Errors

The SPE will terminate and produce an error message if *port_id* is not valid.

Report Variables

None.

SPE_PROBE()

spe_probe(): Checks whether a message on any input port is ready to be received (nonblocking).

Synopsis

```
#include <spe.h>

long spe_probe(void);
```

Return Value

If a message is ready to be received on any input port, then **spe_probe()** returns its port ID. Otherwise, **spe_probe()** returns the predefined constant **SPE_NULL_PORT**.

Description

The **spe_probe()** routine checks if a message on any input port is ready to be received and returns immediately (nonblocking). If a message is ready to be received, then its port ID is returned. Otherwise, **spe_probe()** returns the predefined constant **SPE_NULL_PORT**. See also **spe_wait()** and **spe_probe_list()**.

Each instance of a program must call **spe_probe()**.

Restrictions

spe_probe() cannot be used if there are any connected round-robin control ports in the program. If the program has any round-robin ports, **spe_probe_list()** may be used to probe non-round-robin ports.

Errors

The SPE will terminate and produce an error message if there are no input ports specified in the System Definition file for the calling program.

Report Variables

spe_probe

SPE_PROBE_LIST()

spe_probe_list(): Checks whether a message on one of the specified input ports is ready to be received (nonblocking).

Synopsis

```
#include <spe.h>

long spe_probe_list(
        long        port_id_list[],
        long        num_port_ids);
```

Parameters

port_id_list is a list of port IDs that **spe_probe_list()** will check for a message ready to be received. **spe_probe_list()** will ignore entries in the port ID list that are set to the predefined constant **SPE_NULL_PORT**. The port ID list must contain a least one non-null entry. The port ID list cannot contain port IDs for round-robin control ports.

num_port_ids is the number of port IDs (including any null entries) in *port_id_list*. *num_port_ids* must be greater than zero.

Return Value

If a message is ready to be received on one of the ports specified in *port_id_list*, then **spe_probe_list()** returns its port ID. Otherwise, **spe_probe_list()** returns the predefined constant **SPE_NULL_PORT.**

Description

The **spe_probe_list()** routine checks if a message on one of the input ports specified in *port_id_list* is ready to be received. **spe_probe_list()** immediately returns (nonblocking). If a message is available then its port ID is returned; otherwise, the predefined constant **SPE_NULL_PORT** is returned. **spe_probe_list()** will ignore entries in the port ID list that are set to **SPE_NULL_PORT**. All instances of a program are guaranteed to receive input messages in the same order. See also **spe_msg_wait()** and **spe_probe()**.

Each instance of a program must call **spe_probe_list()** with the same arguments.

Restrictions

portprobe_list() cannot be used to check for a message on a round-robin control port.

Errors

The SPE will terminate and produce an error message if the input port ID list contains a nonvalid input port ID or if it does not specify at least one non-null, connected input port.

Report Variables

spe_probe_list

Example

```
/* Example of how to use spe_probe_list() */
...
data = spe_port_id("data");
cmd1 = spe_port_id("cmd1");
cmd2 = spe_port_id("cmd2");

list[0] = data;
list[1] = cmd2;
...

/* Check for message on any port */
pid = spe_probe()
if (pid != SPE_NULL_PORT)
{
    /* process data */
}

/* Wait for message on the "data" or "cmd2" port */
pid = spe_msg_wait_list(list, 2);
if (pid != SPE_NULL_PORT)
{
    /* process data */
}
```

SPE_PROGRAM_INFO()

spe_program_info(): Copies the program information to the address supplied by the caller.

Synopsis

```
#include <spe.h>

void spe_program_info(
        SPE_PROGRAM_INFO    *prog_info);

typedef struct SPE_PROGRAM_INFO {
        char            name[32];
        long            num_ports;
        long            num_instances;
        long            my_instance;
} SPE_PROGRAM_INFO;
```

Parameters

prog_info is the address to which the program information will be copied.

Description

Copies program information to the address supplied by the caller. The calling program can use this to determine its symbolic name, how many ports it has, how many instances of the program there are, and which instance it is.

Errors

None.

Report Variables

None.

SPE_PROGRAM_SYNC()

spe_program_sync(): Synchronizes all instances of a program.

Synopsis

```
#include <spe.h>

void spe_program_sync(void);
```

Description

Blocks an instance of a program until all instances have arrived at the same point, then returns.

Errors

None.

Report Variables

spe_program_sync

SPE_RECV()

spe_recv(): Posts a receive for a message on an input port and blocks the calling process until the receive completes.

Synopsis

```
#include <spe.h>

void spe_recv(
        long            port_id,
        void            *buf,
        size_t          len,
        SPE_STATUS      *status);

typedef struct SPE_STATUS {
        BOOLEAN     eos;
        long        rows;
        long        columns;
} SPE_STATUS;
```

Parameters

port_id	is the port ID of the input port on which the message will be received.
buf	points to the buffer where the message will be received.
len	is the size of the receiving buffer in bytes. This is used as a consistency check by the SPE. The Program Definition File specifies what the message size should be. If the values do not agree, then the SPE will terminate the run and produce an error message.
status	points to a user-provided buffer where the status information will be copied. If *status* is NULL, then no status is returned. The status information returned indicates whether an EOS mark has been detected.

Description

Posts a receive for a message on an input port and blocks the calling process until the receive completes.

This routine can also detect whether the sender has sent an EOS mark. If an EOS mark is detected, then the *eos* field of the *status* argument will be set to TRUE. If the input port is a striped or replicated port, then the *rows* and *columns* fields will indicate the portion of data that is valid (valid data are contained within rows 0 to *rows – 1* and columns 0 to *columns – 1*, while the remaining portion of the buffer will be filled with zeros). If the port is a control port, then the *rows* and *columns* fields are not used (they are set to zero by the SPE), and the data are not valid. See also **spe_recv_window**().

When EOS is detected and *columns* is non-zero and equal to the input port overlap, then the EOS mark is equivalently at the end of the previously received buffer. In this case, the programmer must make a decision as to whether to process the data in the current buffer.

Unless *port_id* refers to a control round-robin port, each instance of a program must call **spe_recv()** with the same arguments.

Errors

The SPE will terminate and produce an error message if the *len* argument is not the right size or if the *port_id* argument is not a valid input port ID.

Report Variables

spe_recv
spe_eos

SPE_RECV_WINDOW()

spe_recv_window(): Like **spe_recv()** except the SPE uses a window copy to put the data in the user's buffer.

Synopsis

```
#include <spe.h>

void spe_recv_window(
        long            port_id,
        void           *buf,
        size_t          len,
        long            start_column,
        long            num_columns,
        SPE_STATUS     *status);

typedef struct SPE_STATUS {
        BOOLEAN         eos;
        long            rows;
        long            columns,
} SPE_STATUS;
```

Parameters

port_id	is the port ID of the input port on which the message will be received.
buf	points to the overall user's buffer where the message will be received.
len	is the number of bytes that will be received. This is used as a consistency check by the SPE. The Program Definition File specifies what the message size should be. If the values do not agree, then the SPE will terminate the run and produce an error message.
start_column	is the first column to fill in the user's buffer.
num_columns	is the number of columns in the overall user's buffer.
status	points to a user-provided buffer where the status information will be copied. If *status* is NULL, then no status is returned. The status information returned indicates whether an EOS mark has been detected.

Description

Posts a receive for a message on an input port and blocks the calling process until the receive completes.

The difference between **spe_recv_window()** and **spe_recv()** is that this routine specifies that the SPE is to deliver data to a specific rectangular portion (i.e., *window*) of the user's buffer. This may allow the user to avoid allocating an extra buffer and doing an extra copy of the data.

This routine can also detect whether the sender has sent an EOS mark. If an EOS mark is detected, then the *eos* field of the *status* argument will be set to TRUE. If the input port is a striped or replicated port, then the *rows* and *columns* fields will

indicate the portion of data that is valid (valid data are contain within rows 0 to *rows – 1* and columns 0 to *columns – 1*).

Unless *port_id* refers to a control round-robin port, each instance of a program must call **spe_recv_window()** with the same arguments.

Restrictions

Cannnot be used with control ports.

Errors

The SPE will terminate and produce an error message if the *len* argument is not the right size, if the *start_column* argument is out of range, if the *port_id* argument is not a valid input port ID, or if the indicated port is a control port.

Report Variables

spe_recv_window
spe_eos

SPE_REPORT()

spe_report(): Conditionally writes formatted data to the standard output and the log file.

Synopsis

```
#include <spe.h>

void spe_report(
        const char      *report_flag,
        const char      *format,
        ... );
```

Parameters

report_flag is the name of the global database variable that determines whether this routine will write to standard output. *report_flag* must be 31 characters or less.

format is the format string that controls how the data are written. It is identical to the **printf**() format string.

Description

This routine conditionally writes formatted data to the standard output as well as to the log file. It functions identically to **printf**() except that it has an additional argument, *report_flag*, which determines whether the write will actually occur. *report_flag* is the name of a report variable in the global database. The user creates and manipulates report variables for use by the **spe_report**() routine. The **spe_report**() routine will write information based on the contents of the report variable. The following formula is used to decide whether or not to actually write the argument strings:

```
if ((mode == ON) ||
   ((mode == FRAMES) &&
   (current_frame(port_name) >= start_frame)  &&
   (current_frame(port_name) <= end_frame)))
```

The user creates report variables by specifying them in the Database Startup files. From this file, the user can control the contents of report variables, thus affecting which **spe_report**() calls will output data. For a complete description, see Section 4.4.

When the **spe_report**() routine writes data, it provides a header portion indicating the name of the report variable, the name of the calling program, the instance of the calling program, the physical node number, and the time at which it occurred.

Example

To turn on a report that prints the "interesting" variables computed by instance 0 of the beamformer program, between the times that the "gain" port receives its second and fifth message, the following line would be included in one of the Database Startup files:

```
VAR interesting_vars  FRAMES,gain,2,5  beamformer(0)
```

The beamformer program would have embedded **spe_report()** calls in the program where the "interesting" variables are computed.

```
speed_of_sound = ...
spe_report("interesting_vars",
   "speed_of_sound=%f",speed_of_sound);
...
num_bad_sensors = ...
spe_report("interesting_vars",
   "num_bad_sensors=%d", num_sensors);
```

The standard output might look like:

```
REPORT:beamformer(0):gain:frame=2,interesting_vars,
clk=87.887,node=24
-----------------------------------------------------------------
speed_of_sound=1588.1

REPORT:beamformer(0):gain:frame=2,interesting_vars,
clk=87.889,node=24
-----------------------------------------------------------------
num_bad_sensors=0
```

Predefined Report Variables

The variables "error",, "warning", and "info" are predefined report variables that are always ON. When a program uses them in a **spe_report()** call, it forces the formatted data to be written to standard output. When the "error" and "warning" variables are used, an additional large banner is printed to attract the user's attention. The SPE keeps track of how many "error" and "warning" **spe_report()** calls are made and prints a summary at the end of execution. Example usage:

```
spe_report("info","Beginning Initialization");
...
if (speed_of_sound > 2000)
   /* Force this warning to be printed */
   spe_report("warning","speed of sound out of range");
...
```

SPE_REPORT_ENABLED()

spe_report_enabled(): Returns a Boolean value indicating whether a report variable is set so that it would cause a **spe_report**() call to generate output.

Synopsis

```
#include <spe.h>

BOOLEAN spe_report_enabled(
        const char      *report_flag);
```

Parameters

report_flag is the name of the global database variable that determines whether a **spe_report**() routine would generate output. *report_flag* must be 31 characters or less.

Description

This routine returns a Boolean value indicating whether the *report_flag* variable is set so that it would cause a **spe_report**() call to generate output. *report_flag* is the name of a report variable in the global database. The routine is the same routine that the **spe_report**() routine uses internally and is provided so that users can avoid computations that are only used to create optional output information. See **spe_report**() for more information about its usage.

Errors

None.

Report Variables

None.

SPE_SEND()

spe_send(): Sends a message to an output port and blocks until the send completes.

Synopsis

```
#include <spe.h>

void spe_send(
        long        port_id,
        void        *buf,
        size_t      len);
```

Parameters

port_id	is the port ID of the output port to which the message will be sent.
buf	points to the buffer containing the message to send.
len	is the size of the sending buffer in bytes. This is used as a consistency check by the SPE. The port map specifies what the message size should be. If the values do not agree, the SPE will terminate the run and produce an error message.

Description

Sends a message to an output port and blocks until the send completes. When the routine returns, the buffer can be reused. See also **spe_send_window**().

Each instance must call **spe_send**() when sending data to a striped, replicated, or regular control port. For replicated or regular control ports, the data sent must be the same on each instance (the SPE will determine, based on efficiency, which instance(s) will actually transmit the data).

Errors

The SPE will terminate and produce an error message if the *len* argument is not the right size or if the *port_id* argument is not a valid output port ID.

Report Variables

spe_send

SPE_SEND_WINDOW()

spe_send_window(): Like **spe_send**() except the SPE uses a window copy to get the data from the user's buffer.

Synopsis

```
#include <spe.h>

void spe_send_window(
        long        port_id,
        void        *buf,
        size_t      len,
        long        start_column,
        long        num_columns);
```

Parameters

port_id	is the port ID of the output port to which the message will be sent.
buf	points to the overall user's buffer containing the message to send.
len	is the number of bytes of data to send. This is used as a consistency check by the SPE. The port map specifies what the message size should be. If the values do not agree, the SPE will terminate the run and produce an error message.
start_column	is the first column of the user's buffer from which to get data.
num_columns	is the number of columns in the overall user's buffer.

Description

Sends a message to an output port and blocks until the send completes. When the routine returns, the buffer can be reused.

The difference between **spe_send_window**() and **spe_send**() is that this routine specifies that the SPE is to send data from a specific rectangular portion (i.e., *window*) of the user's buffer. This may allow the user to avoid allocating an extra buffer and doing an extra copy of the data.

Each instance must call **spe_send_window**() when sending data to a striped, replicated, or regular control port. For replicated ports, the data sent must be the same on each instance (the SPE will determine, based on efficiency, which instance(s) will actually transmit the data).

Restrictions

Cannnot be used with control ports.

Errors

The SPE will terminate and produce an error message if the *len* argument is not the right size, if the *port_id* argument is not a valid output port ID, or if the indicated port is a control port.

Report Variables

spe_send_window

SPE_TERMINATE()

spe_terminate(): Tells the SPE to terminate the application.

Synopsis

```
#include <spe.h>

void spe_terminate(void);
```

Description

Tells the SPE to terminate the application. The SPE will cause each program in the system to terminate when the next SPE routine is called, or if already in an SPE routine, to terminate immediately. (It does not interrupt what the user's program is currently doing.) Each program will execute an optionally defined user termination routine (see **spe_terminate_define**()), will generate any **spe_report**() summaries that have been requested, and will then exit. Any program in the system can initiate a system termination by calling this routine.

Errors

None.

Report Variables

spe_terminate

SPE_TERMINATE_DEFINE()

spe_terminate_define(): Specifies a function to be executed when the program terminates.

Synopsis

```
#include <spe.h>

void spe_terminate_define(
        void        (*term_function) (void));
```

Parameters

term_function is the name of function to execute when the program terminates. The function must have no arguments and return no value.

Description

Specifies a function to be executed when the program terminates. This allows a program to execute critical cleanup code (such as closing files) when the program is terminated by some other program. See **spe_terminate**().

Errors

None.

Report Variables

None.

Appendix D: FORMAT OF DESCRIPTION FILES

The SPE is controlled by a number of ASCII files: the System Definition file, Program Definition files, and the Database Startup files. This Appendix describes the format and grammar of these files.

COMMENTS

Comments may be included freely to describe and document the contents of configuration files. A comment begins with a double forward slash (//). This symbol and the rest of the line are ignored by SPE processing.

C PREPROCESSOR DIRECTIVES

Before processing the configuration files, the SPE filters them through the C language preprocessor `cpp`. This allows the user to take advantage of all features of the preprocessor such as including other files, defining variables, and defining macros.

GRAMMAR

The grammar of the various configuration files is defined by reserved words and user-provided tokens of type *identifier*, *integer*, *real*, and *string*. Tabs, spaces, and blank lines are white space and are used to delimit lines into tokens interpreted by the parser; they are otherwise ignored.

Identifiers must be an allowable C language identifier, up to 31 characters in length, and cannot be any of the reserved words. Reserved words must be specified either as all upper or all lower case; user identifiers are case sensitive.

Integers must be specified as an integer without a decimal point (e.g., 100).

Real values contain a decimal point (e.g., 1500.0) or an exponent (e.g., 1e+3) or both; their type is internally represented as a double.

Strings are enclosed in double quotes (e.g., `"sea_test1"`) and are limited to 254 characters in length.

EXPRESSIONS

The SPE allows integer, real, and string-valued expressions. Precedence and associativity of operators allowed in expressions are shown in the following table (listed from high to low precedence).

Operators	Associativity
unary_minus `(int)` `(double)`	right to left
`*` `/` `%`	left to right
`+` `-`	left to right
`&`	left to right
`|`	left to right

Some notes on expressions:

1. Real expressions can mix real and integer numbers. Binary operations combining real and integer numbers will promote the result to a real value.

2. The `(int)` operation converts a real or integer expression to an integer number. The `(real)` operation converts a real or integer expression to a real number.

3. The '+' operation concatenates strings.

GRAMMAR

Each command of an SPE configuration file begins with a reserved word and is terminated by the end of the line. A "logical" line may be extended to multiple actual lines by ending intermediate lines with a backslash character (\).

Reserved words and symbols are shown below in `typewriter` font. The symbol (...) means that the previous symbol may be repeated. The symbol (,...) means that the previous symbol may be repeated, but if so, the symbols must be separated by a comma (,). Alternatives are show below on separate lines.

System_Definition_File:
system_statement . . .

system_statement:
buffer_statement
dump_statement
exclude_statement
net_statement
program_statement
transpose_statement

Program_Definition_File:
port_statement . . .

Database_Startup_File:
variable_statement . . .

buffer_statement:
`buffer` *net_connection integer_expr*

dump_statement:
`dump` *net_connection dump_array_spec* `matlab =` *dump_format dump_option* . . .
`dump` *net_connection dump_array_spec* `ascii =` *dump_format dump_option* . . .
`dump` *net_connection dump_array_spec* `spe =` *dump_format dump_option* . . .

exclude_statement:
`exclude` *program_name*

net_statement:
`net` *net_connection*`,`*net_connection* `,` . . .

program_statement:
`program` *node_specification program_name program_def_path program_exec_path*

transpose_statement:
`transpose` *net_connection*

dump_array_spec:
dump_row_spec dump_column_spec

dump_row_spec, dump_column_spec:

```
[ : ]
[ integer_expr : ]
[ : integer_expr ]
[ integer_expr : integer_expr ]
```

dump_format:

```
"double"
"double_complex"
"float"
"float_complex"
"int"
"int_complex"
"short"
"short_complex"
"ushort"
"ushort_complex"
"uchar"
"uchar_complex"
```

dump_option:

```
append
convert
convert = convert_flag
filename = string_expr
frames = start_frame : end_frame
frames = one_frame
label = string_expr
no_header
rename = string_expr (this string limited to 31 characters or fewer)
structure = dump_offset , dump_elements
structure = dump_offset , dump_elements , dump_stride
structure = dump_offset , all
structure = dump_offset , all , dump_stride
```

net_connection:

```
program_name : port_name
```

node_specification:

```
number_of_instances
( min_nodes , max_nodes , node_weight )
```

port_statement:

```
port port_name input  control
port port_name input  control sequence
port port_name output control
port port_name output control round_robin
port port_name input  replicated [ rows ] [ columns ] elem_size bovlp
port port_name output replicated [ rows ] [ column ] elem_size
port port_name input  striped [ rows ] [ columns ] elem_size sovlp bovlp
port port_name output striped [ rows ] [ columns ] elem_size
```

rows, columns, elem_size:
   any_expr

any_expr:
   `any`
   integer_expr

bovlp:
   / empty */*
   `block_ovlp =` *integer_expr*

sovlp:
   / empty */*
   `striped_ovlp =` *ovlp*
   `striped_ovlp =` *start_ovlp* `:` *end_ovlp*
   `striped_ovlp =` *ovlp* `: all`
   `striped_ovlp =` *start_ovlp* `:` *end_ovlp* `: all`

variable_statement:
   `var` *variable_name db_value db_destination*

db_value:
   integer_expr
   real_expr
   string_expr
   `true`
   `false`
   `off`
   `on`
   `frames` , *port_name* , *start_frame* , *end_frame*

db_destination:
   / empty */*
   program_name
   program_name (instance)

program_name, port_name, variable_name:
   identifier

convert_flag, dump_elements, dump_offset, dump_stride,
end_frame, end_ovlp, instance, max_nodes, min_nodes,
number_of_instances, one_frame, ovlp, start_frame, start_ovlp:
   integer_expr

node_weight:
   real_expr

program_def_path, program_exec_path, dump_fmt_descriptor:
   string_expr

real_expr:
- *(real_expr)*
- *real_expr + real_expr*
- *real_expr – real_expr*
- *real_expr * real_expr*
- *real_expr / real_expr*
- *– real_expr*
- `(real)` *integer_expr*
- *real_function*
- *real*

integer_expr:
- *(integer_expr)*
- *integer_expr + integer_expr*
- *integer_expr * integer_expr*
- *integer_expr / integer_expr*
- *integer_expr % integer_expr*
- *integer_expr & integer_expr*
- `(int)` *real_expr*
- *integer_function*
- *integer*

string_expr:
- *string_expr + string_expr*
- *string*

integer_function:
- `ceil` *(real_expr)*
- `floor` *(real_expr)*

real_function:
- `max` *(real_expr , real_expr)*
- `min` *(real_expr , real_expr)*

Appendix E: STRIPE AND OVERLAP ALGORITHMS

BASIC STRIPE ALGORITHM

The basic striping algorithm used to compute the range of rows that will be allocated to each instance of a program is as follows:

```
if (instance < num_rows % num_instances)
{
   start_row = instance * (num_rows / num_instances + 1);
   end_row   = start_row + (num_rows / num_instances);
}
else
{
   start_row = instance * (num_rows / num_instances) +
               (num_rows % num_instances)
   end_row   = start_row +  (num_rows / num_instances)– 1
}
```

For example, if an array with 100 rows is striped over 3 instances, then instance 0 will have rows 0 to 33, instance 1 will have rows 34 to 66, and instance 2 will have rows 67 to 99. Note that the first instance has one more row than the others. This is illustrated in figure E-1. The columns dimension of the array does not affect the striping.

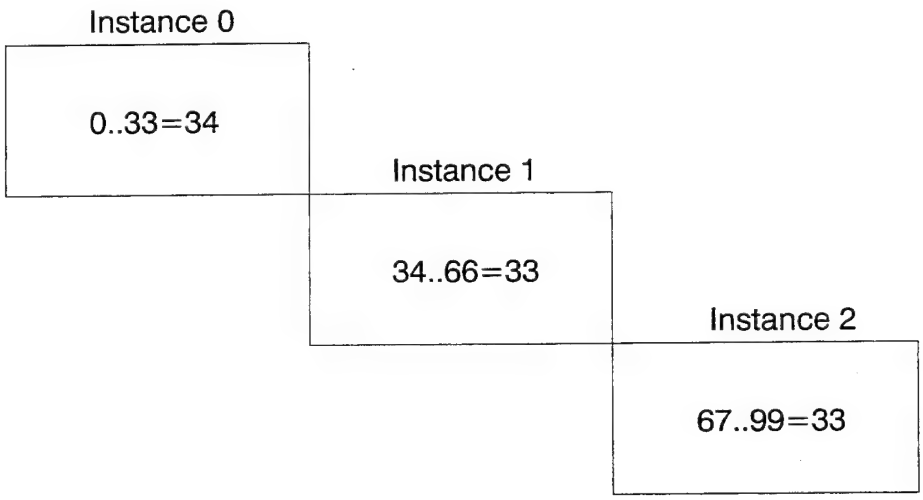


Figure E-1. Row allocation for basic striping with no overlap.

STRIPE OVERLAP ALGORITHMS

The SPE allows the user to specify for an input port that the incoming data rows should be distributed to the program instances with an *overlap* of rows between adjacent instances. In general terms, the overlap is specified by giving the number of rows of overlap needed by the application. The overlap can be *symmetrical* or *asymmetrical*. That is, considering the block of rows allocated to a particular program instance, the number of rows of overlap at the beginning of a block can be the same (symmetrical) or different (asymmetrical) than the rows of overlap at the end of the block. In addition, the SPE allows the overlap on the first and last instances of a program to be treated

differently than on the other instances, a feature required by some applications. There are four forms of overlap specification (see Appendix D for a description of the syntax). Each case is illustrated for the same example used above in which 100 input rows are allocated to three program instances. The figures indicate graphically the allocation of the rows to instances; the overlapped rows are shaded and annotations within the figures show the specific row numbers and total number of rows for each instance.

STRIPED_OVLP=*ovlp*

In this, the simplest case, a single value is given for a symmetrical overlap. The *ovlp* argument gives the number of extra rows to be sent to a program instance at both the start and end of the block. This is in addition to the block of rows that would have been allocated by the basic algorithm. Figure E-2 illustrates this case. Note that the first and last instances are allocated a smaller number of rows because there is overlap on only one edge. The assumption here is that the application itself does something special at the outside edges (rows) of the overall data block.

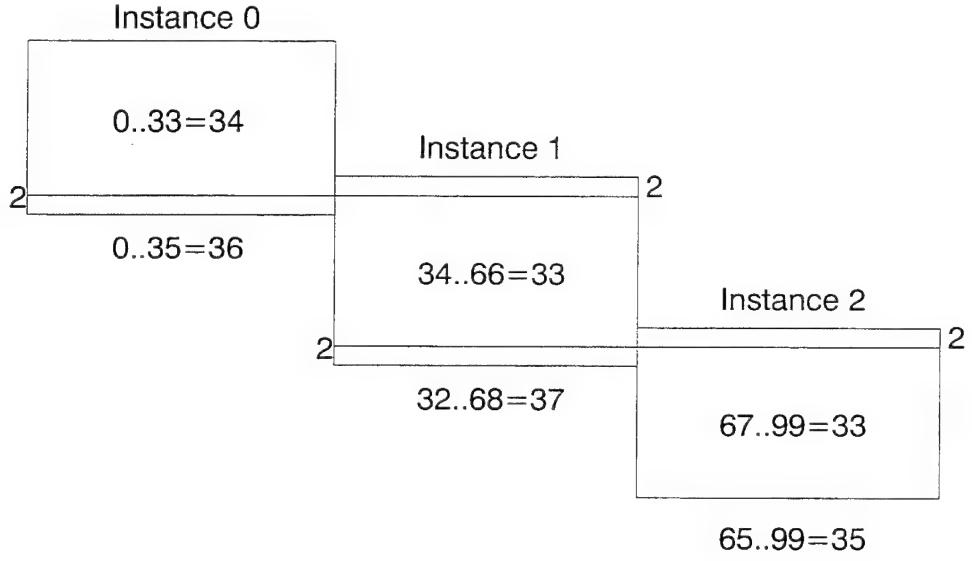


Figure E-2. Row allocation for `STRIPED_OVLP=2`.

STRIPED_OVLP=*start_ovlp*:*end_ovlp*

In this case, two overlap values are given to describe an asymmetrical overlap. The *start_ovlp* argument gives the number of extra rows at the start of the block, and the *end_ovlp* argument gives the number of extra rows at the end of the block. This is in addition to the block of rows that would have been allocated by the basic algorithm. Figure E-3 illustrates this case. As with the symmetrical case of figure E-2, the first and last instances are allocated a smaller number of rows on the assumption that the algorithm itself does something special at the outside edges (rows) of the overall data block.

STRIPED_OVLP=*ovlp*:ALL

This case differs from the simple case of figure E-2 in that the application requires an overlap at the outside edges of the overall data block; that is, the first and last program instances must have the same overlap as the other instances. This is done by executing the basic allocation algorithm with a smaller number of rows: the total rows in the overall data block minus twice the amount of the *ovlp*

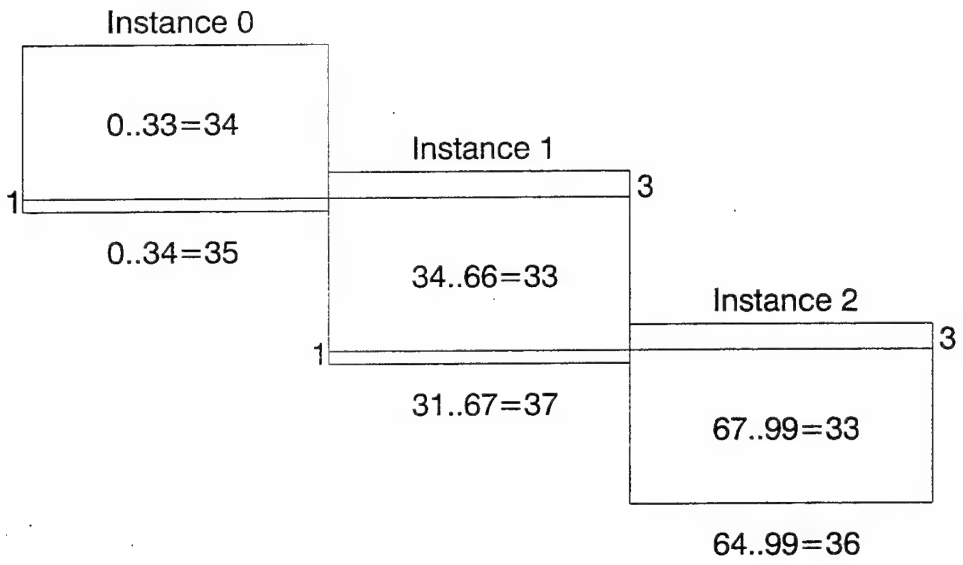


Figure E-3. Row allocation for `STRIPED_OVLP=3:1`.

argument. Then when the overlap is allocated uniformly to every instance, the first and last instances have the same overlap to work with as all other instances. Figure E-4 illustrates this case. Note that the overall program will now output a fewer number of rows than was input. In the example, only 96 rows will be output.

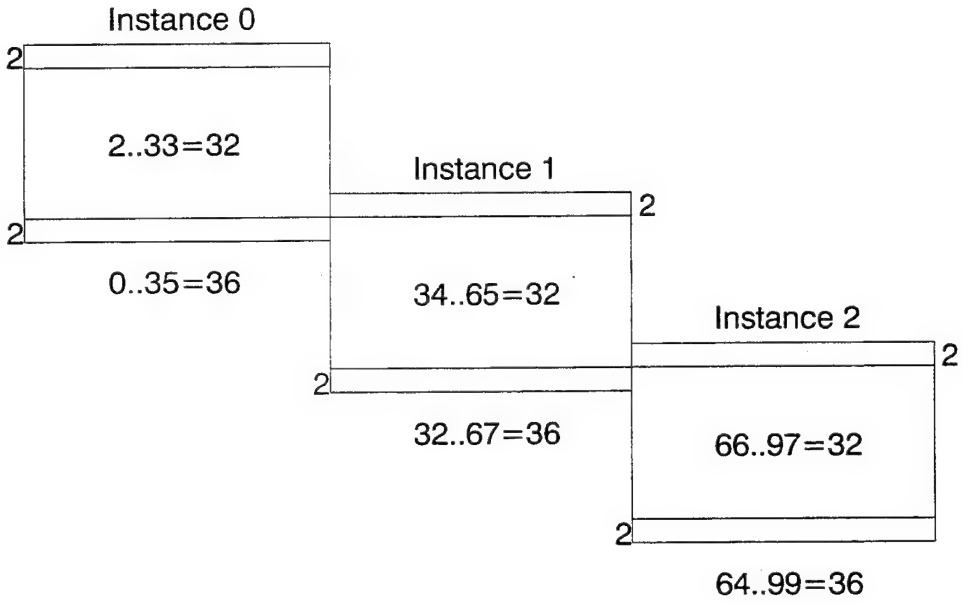


Figure E-4. Row allocation for `STRIPED_OVLP=2:ALL`.

`STRIPED_OVLP=`*start_ovlp*`:`*end_ovlp*`:ALL`

This is the same as the case in figure E-4 except that it allows for asymmetrical overlap as shown in figure E-3. Here again the result is that the first and last instances have the same overlap to work with as all other instances. Note as before that only 96 rows are output from the program. Figure E-5 illustrates this case.

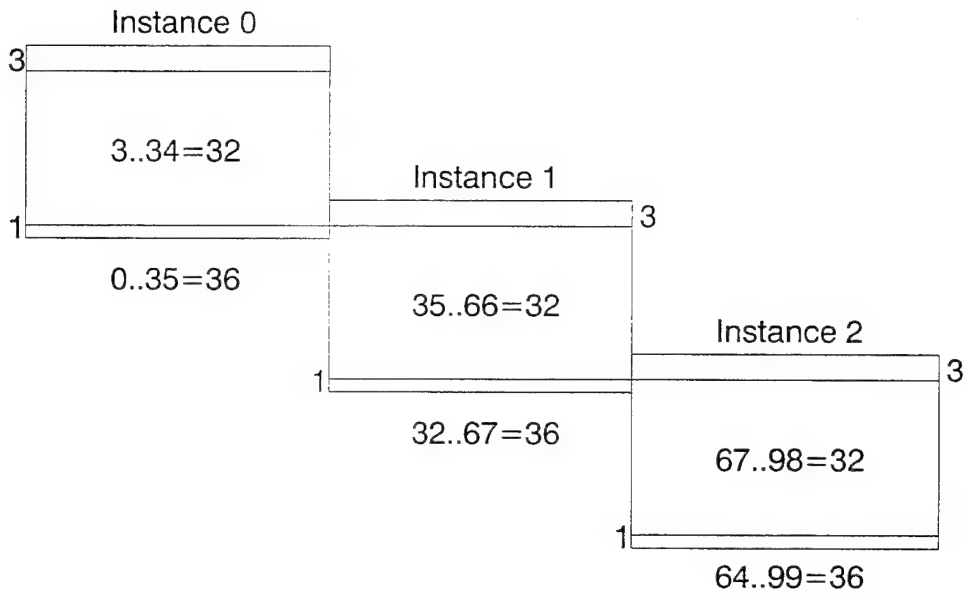


Figure E-5. Row allocation for `STRIPED_OVLP=3:1:ALL`.

INDEX

Reserved words are shown here in all capitals, e.g., EXCLUDE. *Function names are given with appended parentheses, e.g.,* spe_report().

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1995	Final: FY 1994–FY1995

4. TITLE AND SUBTITLE
SCALABLE PROGRAMMING ENVIRONMENT

5. FUNDING NUMBERS
PE: 0602314N
PROJ: RJ14C31
SUBPROJ: SUBP 01
ACC: DN 302019

6. AUTHOR(S)
Perry Partow and Dennis Cottel

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Command, Control and Ocean Surveillance Center (NCCOSC)
RDT&E Division
San Diego, CA 92152–5001

8. PERFORMING ORGANIZATION REPORT NUMBER
TR 1672, Rev. 1

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217–5660

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report describes the Scalable Programming Environment (SPE), which provides programmers with a transparent way of creating scalable parallel applications for large-grained parallel computer architectures. The SPE, which has been designed primarily to support data-flow processing applications, allows programs to be scaled to execute on any number of processing nodes while requiring no changes to the compiled binary code. The user is provided with a set of high-level message-passing routines that can be used to connect multi-instanced heterogeneous programs in a system. The SPE library routines hide the intricacies of how the parallel programs communicate. The details of the connections are specified in text files. The SPE allows individual programs to be coded without knowledge of other parts of the system and thus allows systems to be quickly built, modified, or scaled without program recompilation.

14. SUBJECT TERMS
parallel dataflow
parallel message passing
parallel programming tools
parallel processing
scalable
scalable programming environment

15. NUMBER OF PAGES
114

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAME AS REPORT

21a. NAME OF RESPONSIBLE INDIVIDUAL	21b. TELEPHONE *(include Area Code)*	21c. OFFICE SYMBOL
Perry Partow	(619) 553–1663	Code 782

INITIAL DISTRIBUTION

Code 0012	Patent Counsel	(1)
Code 0275	Archive/Stock	(6)
Code 0274	Library	(2)
Code 70	T. F. Ball	(1)
Code 78	P. M. Reeves	(1)
Code 782	R. A. Dukelow	(1)
Code 782	D. M. Cottel	(1)
Code 782	P. P. Partow	(50)
Code 784	J. C. Lockwood	(1)

Defense Technical Information Center
Fort Belvoir, VA 22060–6218 (4)

NCCOSC Washington Liaison Office
Washington, DC 20363–5100

Center for Naval Analyses
Alexandria, VA 22302–0268

Navy Acquisition, Research and Development
Information Center (NARDIC)
Arlington, VA 22244–5114

GIDEP Operations Center
Corona, CA 91718–8000

Office of Naval Research
Arlington, VA 22217–5000